1991

Findings of a comparison of five filing protocols

R. Elayne McFaul

Rochester Institute of Technology

School of Computer Science and Technology

Findings of a Comparison of Five Filing Protocols

May 1991

R. Elayne McFaul

A thesis, submitted to the Faculty of the School of Computer Science and Technology, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Approved by:

Susan M. Armstrong

Peter A. Crean

James Heliotis

Charles H. Russell

I, Elayne McFaul, prefer to be contacted each time a request for reproduction of this thesis is made. I can be reached in one of the following ways:

Xerox Corporation
800 Phillips Road 128-53E
Webster, NY 14580

716-422-4328

mcfaul.wbst128@xerox.com

Table of Contents

List of Figures

Abstract

Filing protocols are essential for the management and dissemination of shared information within computer systems. This is a survey of the current state of the art in filing protocols. Five popular filing protocols were selected and subjected to a rigorous comparison. FTAM, FTP, UNIX rcp, XNS Filing, and NFS are compared in the following areas: exported interface, concurrency control, access control, error recovery, and performance. The coverage of background material includes a taxonomy and a brief history of filing protocols.

Keywords and Phrases

Access Control, Concurrency Control, Error Recovery, Exported Interfaces, Filing, Filing Systems, FTAM, FTP, Network Protocols, NFS, Performance, Sun Microsystems, UNIX, UNIX rcp, XNS.

Computing Review Subject Codes

C.2.2 Computer Systems Organization: [Computer-Communication Networks]: Network Protocols

C.2.4 Computer Systems Organization: [Computer-Communication Networks]: Distributed Systems

C.4 Computer Systems Organization: [Performance of Systems]

E.5 Data: [Files]

1.0 Introduction

Over the past decade, computer networks have become widespread, linking together a diverse variety of systems and their users. It has become common to exploit such network interconnections to distribute software, access remote resources (such as hardware and databases), operate diskless workstations, send electronic mail and conduct computer conferences. Fundamental to these activities is the need for a means of handling the information that is to be shared or exchanged between the systems.

To meet this demand, several organizations and vendors have designed their own "standards" for transferring units of information, known as *files*. File transfer means copying or moving an entire file or a portion of a file from one machine to another, and it is one of the most frequently used network operations. A file transfer operation requires cooperation between at least two systems, which must follow a set of mutually agreed-on rules, or a *protocol*. Filing protocols have increased the resources available to networked computer users by potentially making the file resources of every computer on the network available to any other user on the network.

Heterogeneity is a part of networking. As a result, the computers in a given network may very well be running several operating systems on different hardware platforms. Unfortunately, different operating systems usually have different file systems. In some cases the differences are relatively simple, such as the naming conventions for files. In other cases the differences are dramatic. For example, the UNIX operating system has a simple view of a file: a stream of bytes that can be accessed arbitrarily by offset. In contrast, other operating systems support structured files, in which access is key-based for storing, manipulating and transferring files.

The amount of network traffic a file transfer generates is related to the protocol followed to perform the transfer, the number of bytes in the file and, of course, the underlying transport, network and data link protocols. The advent of image processing, graphics and video applications is dramatically enlarging the size of an average file and increasing the importance of selecting a protocol best suited to perform the transfer.

Making comparisons between protocols is a notoriously difficult problem. No two protocols have the same set of features nor do they achieve the same functionality. Filing protocols are typically defined entirely within the application layer, or layer 7 of the Open Systems Interconnection (OSI) Reference Model, shown in Figure 1. A true comparison of any type of

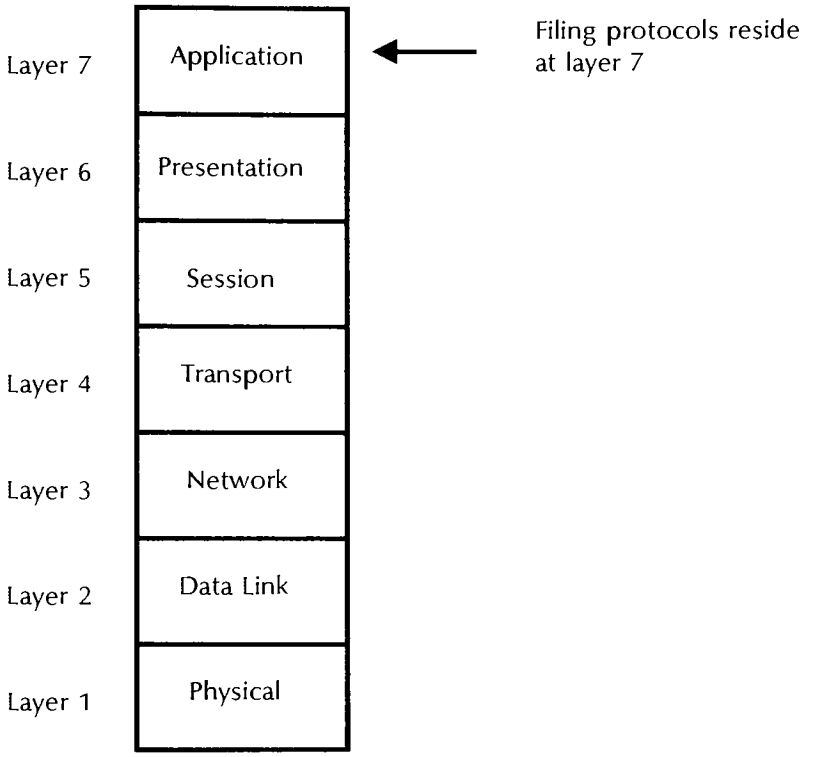


Figure 1. OSI 7-layer model

application-level protocol is further complicated by differing underlying protocol stacks (layers 1-6).

In this paper, a multi-dimensional comparison approach was selected. A study of all the formal protocol specifications was done. Then, using the protocol specifications and reference implementations, a comparison in five key protocol design areas was performed:

- exported interface
- concurrency control
- access control

- error recovery
- performance

Unfortunately, file transfer performance can be influenced by many variables unrelated to the actual filing protocol. Every effort was made to minimize these variables. Frequently, a filing protocol's performance is only as good as its underlying transport protocol implementation (layer 4). Because the paper's focus is the comparison of filing protocols, not transport protocols, a common transport protocol, the Transmission Control Protocol (TCP) was selected. Where feasible, those filing protocols that do not use a TCP transport were modified to use TCP.

This document attempts to describe the rationale surrounding the chosen protocol comparison method and reports the results of the comparison. Section 1.1 describes the literature review. Section 1.2 outlines the goal of the thesis. An introduction to each of the protocols is made in section 2. Section 3 contains the descriptions of the five key design areas for each protocol. Section 4 draws conclusions from the study. Section 5 makes acknowledgements and the references appear in Section 6. Appendix A contains a comprehensive table-formatted comparison of the five protocols. This version of the table pulls together all the comparison tables from the body of the document. Appendix B contains C, UNIX, and SAS source code listings relating to the performance measurements. Appendix C contains C source code relating to the porting of XNS Filing to a TCP transport.

1.1 Literature Review

Research in the design of filing protocols dates back to the mid 1970s with Xerox's PUP (PARC Universal Packet) File Transfer Protocol [Xerox75] and the DoD's File Transfer Protocol. Since then, many filing protocols have been designed and implemented.

There are two different classes of filing protocols: those defining a *general-purpose file transfer system*, and those defining a *network file system*.

- *General-purpose file transfer protocols* are used for copying and accessing files. They do not define the file systems of their computer hosts. Implementations of this class of protocols typically reside as application programs and the interface is usually interactive or through a spooling system. Examples of this protocol class are FTAM, FTP, UNIX rcp, and XNS Filing.

- *Network File System protocols* try to seamlessly extend a computer's file system across a network. In order to do this transparently, the local operating system needs to support this functionality, and the interface is sometimes inside the operating system. Examples of this protocol class are Sun Microsystem's Network File System (NFS), AT&T's Remote File Sharing (RFS), Hewlett-Packard/Apollo Computer's DOMAIN, and Carnegie Mellon University's Andrew File System (AFS).

A graphic illustration of the two classes of filing protocols appears in Figure 2.

Five filing protocols were selected for comparison work in this thesis. The general-purpose file transfer class is represented by ISO's File Transfer, Access and Management (FTAM), FTP, the UNIX remote copy utility rcp, and Xerox Network System (XNS) Filing. Sun's NFS rounds out the comparison by representing the network file system protocol class. Although NFS is based on a substantially different model than the other four protocols, its popularity is too widespread to exclude it from this study. There are many reasons why NFS is more popular than other network file systems: it was designed for a network of mixed protocols, mixed machine types, and mixed operating systems, its performance is good, and it is available today.

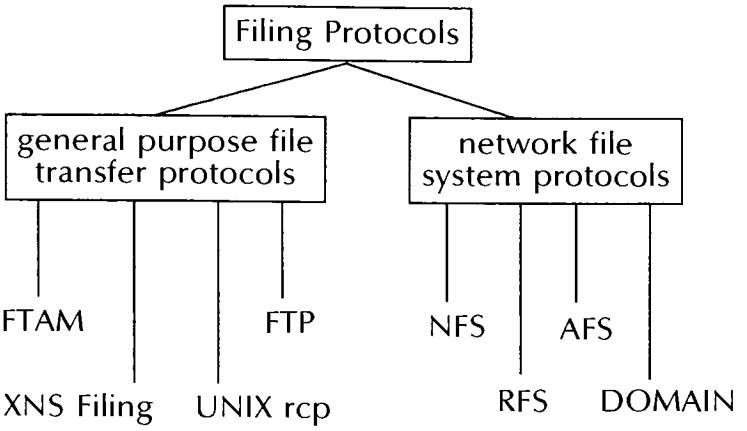


Figure 2. Taxonomy of Filing Protocols

Common to all filing protocols is the model of a client/server relationship. A *client* is an entity requesting work to be done and a *server* is an entity accepting request for work. An entity is a process running on either a multi-process computer or a dedicated computer.

Another categorization distinguishes those filing protocols which are based on remote procedure calls and those that are asynchronous message-based. Remote procedure calls do for distributed system builders some of what a high-level programming language does for implementers of non-distributed systems. Just as Pascal allows a system builder to think in terms of procedure calls rather than in terms of base registers and branch and link instructions, distributed system builders can think in terms of remote procedure calls rather than in terms of socket numbers and network connections. A remote procedure call package consists of a library of procedures. When a client directs a server to execute a procedure call, the actual execution takes place in the address space of the server, but it appears to the client as though it was executed in its own address space. Remote procedure calls are generally synchronous, that is, the client application waits until the server has completed the call and returned the results. In this study, XNS Filing and NFS are remote procedure call-based protocols, while FTP, FTAM, and UNIX rcp are asynchronous message-based.

1.2 Thesis Goal Statement

The goal of the thesis is a rigorous comparison of five popular application level file transfer protocols and representative implementations:

1) File Transfer, Access and Management (FTAM), a standard from the International Standards Organization [ISO88a-c].

2) The File Transfer Protocol (FTP), defined by the U.S. Department of Defense [DDN85].

3) UNIX remote copy (rcp), from the University of California at Berkeley.

4) Xerox Network Systems (XNS) Filing, defined by Xerox Corporation [Xerox81a].

5) Network File System (NFS), defined by Sun Microsystems, Inc. [Sun86b].

Comparisons will be made in the following areas: exported interface, concurrency control, access control, error recovery, and performance. Every attempt will be made to run the representative implementations on a common software and hardware platform. The common software platform is TCP with a Berkeley-based socket interface. The common hardware platform is Sun Microsystems' workstations. Only disk to disk transfers will be considered. Figure 3 shows the selected protocols' relative position in the top three layers of the OSI 7-layer model.

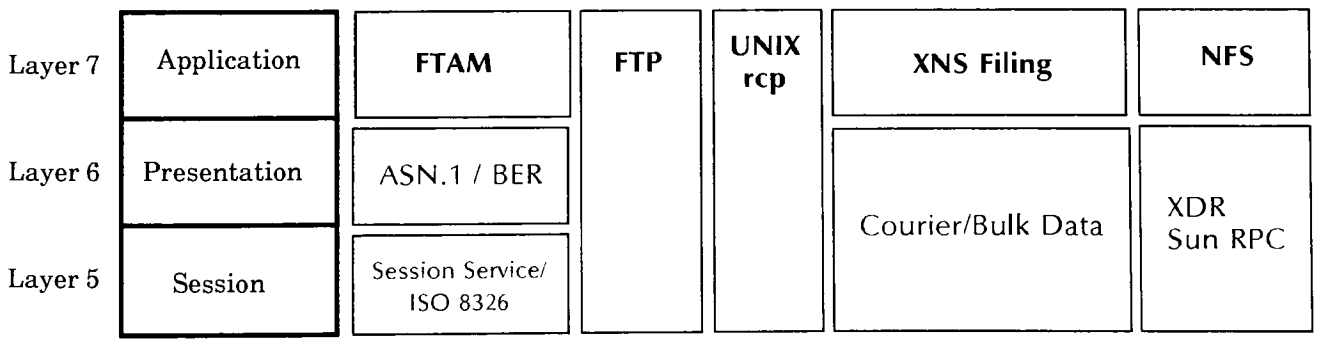


Figure 3. Each protocols' position in layers 5-7 of ISO's 7-layer model

2.0 General Protocol Descriptions

2.1 FTAM

The File Transfer, Access, and Management (FTAM) protocol is a recommendation from the International Standards Organization (ISO) to standardize file transfer, access and management among heterogeneous interconnected computer systems.

Work on FTAM began in the late 1970's, and it became an international standard in 1987. Because many countries have made a commitment to the adoption of OSI, these protocols are expected to play a major role in worldwide communications. In 1990, FTAM became a United States Federal Government procurement requirement. The current ISO FTAM work concerns primarily file transfer, while the more complex part of the access and management features are to be defined in future revisions of the FTAM standard. FTAM is designed to support every computer hardware platform and every software operating system. The goal of the protocol is to permit any two end systems to transfer any type of data between themselves. The transfer is performed without either end system having any prior knowledge of the other end system's configuration.

Different systems have their own peculiar styles of describing the storage of data and the ways in which data can be accessed. The ISO FTAM protocol defines a standard for transferring, accessing, and managing files among open systems without having to know how file storage is implemented on each system. In FTAM, a reference model to promote a universal view of files is adopted and referred to as the virtual filestore (VFS). The VFS is how FTAM views files, as opposed to the real filestore, which is how an operating system views files. A local mapping function can then absorb the style and specification differences between the VFS and the real filestore. From FTAM's perspective, there is no file system directory structure. Each file has a set of attributes and contents. A file is considered to be a tree, and each subtree, or file access data unit (FADU) can potentially be accessed independently. The nodes of the tree may each carry identifiers and may have file data units associated with them.

FTAM provides a confirmed service where both end systems are always in a mutually known state. It uses connection-oriented session and transport protocols. The FTAM protocol is session-oriented. The conceptual model for the FTAM protocol has two main entities: the initiator and the responder. The initiator is analogous to a client, and the responder is analogous to a server. All FTAM implementations must be able to act as either initiator or responder. The initiator submits requests to the responder, who services the requests through the aid of the virtual filestore.

The interactions between the initiator and the responder are governed by a series of four nested regimes: the FTAM regime, the file selection regime, the file open regime and the data transfer regime. Within each regime, a set of permitted operations is maintained. For example, F-OPEN and F-CLOSE are legal operations during the file open regime.

FTAM uses the Association Control Service Element (ACSE) [ISO88d] to manage the association between the initiator and the responder. In addition, FTAM makes use of an abstract syntax language and a transfer syntax language. OSI currently has one abstract syntax language--Abstract Syntax Notation One (ASN.1) and one transfer syntax notation--Basic Encoding Rules (BER). ASN.1 is used to map the data to an abstract syntax, or a machine-independent representation. BER then takes the abstract syntax and maps it to a concrete syntax. The result of this mapping is a stream of octets ready to transmit across the network. Exchange of information is possible as long as the end systems share a uniform view of a set of abstract data elements.

The task of transferring a file via FTAM starts with the following steps: establishing an FTAM session, selecting a file, and opening the file. Once the file is opened, an initial command specifies the direction and the content of the file transfer, followed by the transfer itself, and is completed by an exchange of terminating acknowledgements before the file is closed, deselected, and the session is relinquished. The full sequence of events in transferring a file from one system to another is shown is Figure 4.

The ISO Development Environment (ISODE), pioneered by Marshall T. Rose and Dwight E. Cass, is an openly available implementation of the upper levels of OSI. The ISODE implementation of FTAM uses TP0 over TCP/IP at the transport and network levels. TP0 is the simplest of all the ISO transport

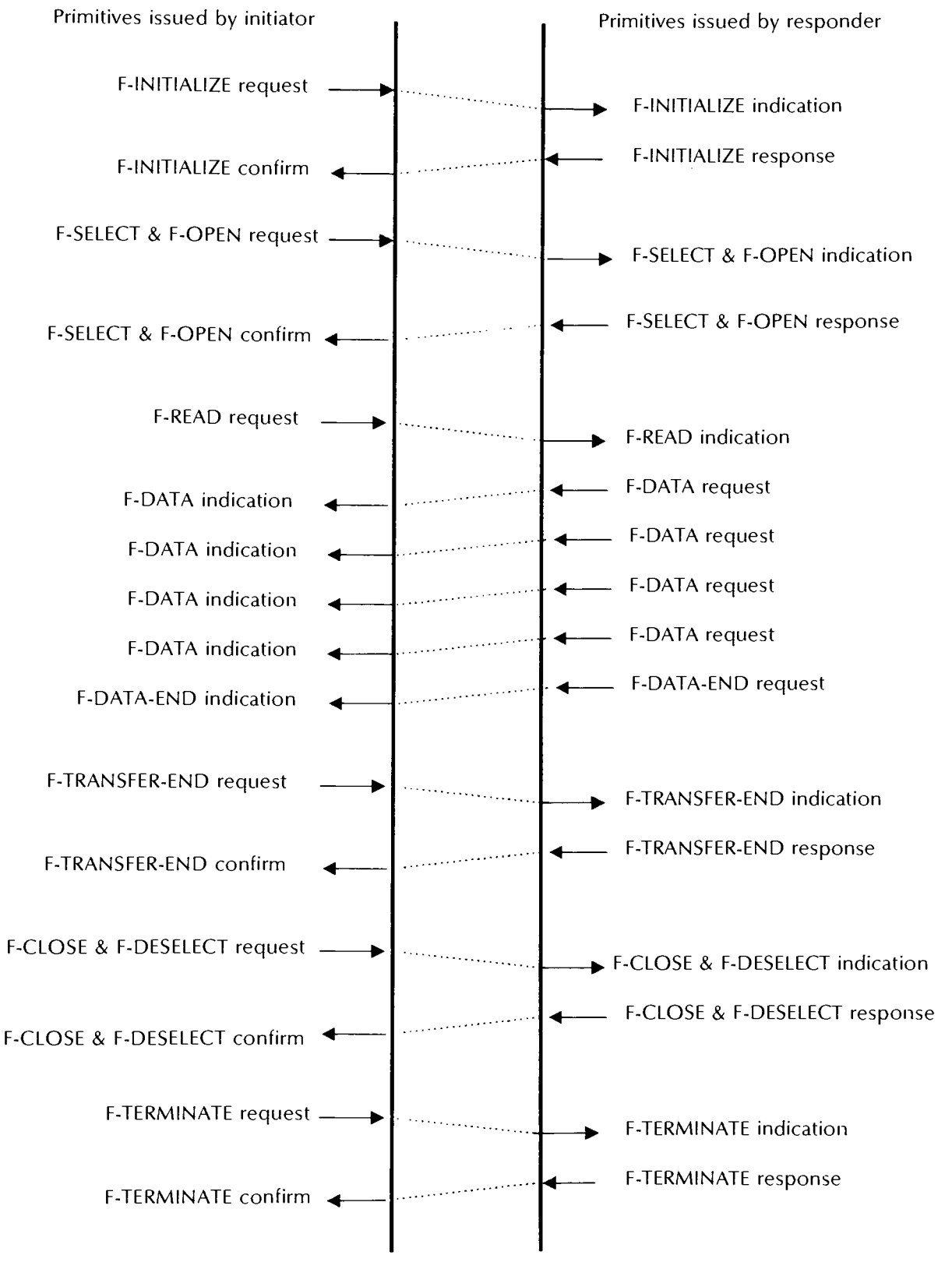


Figure 4. Remote file access using FTAM

protocols, responsible for simple segmentation and reassembly. As a collection of binaries and application programs, the ISODE was designed as a tool to study OSI, but has emerged as the de facto reference implementation of the OSI upper levels, and has become the basis for some OSI production software.

2.2 FTP

In the late 1960s and 1970s, there was only one major national computer network, called the ARPANET, which connected a few dozen computer systems around the country. The ARPANET was sponsored by the Defense Advanced Research Projects Agency (DARPA) of the Department of Defense (DoD). In the early 1980s, a new family of protocols was specified as the standard for the ARPANET and associated DoD networks. This family of protocols is known as the Internet Protocol suite. Because of its heavy dependence on the Transmission Control Protocol (TCP) and the Internet Protocol (IP), the protocol family is also known as the TCP/IP protocol suite, and sometimes simply TCP/IP. TCP is a connection-oriented transport service that provides a confirmed service so that both the client and the server are always in a mutually known state.

One of the protocols in the Internet suite is a file transfer protocol called FTP. FTP is a popular protocol with a mature implementation base. Many implementations of FTP exist, some which offer features and functionality beyond what is defined in the original standard protocol specification. FTP is a session oriented protocol which deals with heterogeneous systems by understanding a few basic file formats, various end-of-line conventions, and several character representations. Unlike most protocols, FTP separates control and bulk data transfer onto two separate connections. FTP was designed to operate in two modes: either by direct interaction with humans, or from a computer program.

A utility program called ftp is the most common user interface to the File Transfer Protocol. It allows authorized users to log into a remote system, identify themselves, list remote directories, copy files to or from the remote machine, and delete files on the remote machine.

A model for a transfer between two machines, the client *C* and the server *S*, is illustrated in Figure 5. A background process runs on *S* that listens for a TCP Telnet connection at a well-known FTP port. A process on *C* opens a TCP connection to this port, presents user credentials, and asks for a file transfer. *S* then initiates a second TCP connection to *C* for the data flow. When all the data has been transferred, *S* closes this data connection. The initial TCP control connection (the Telnet session) is unaffected by the closure of the data

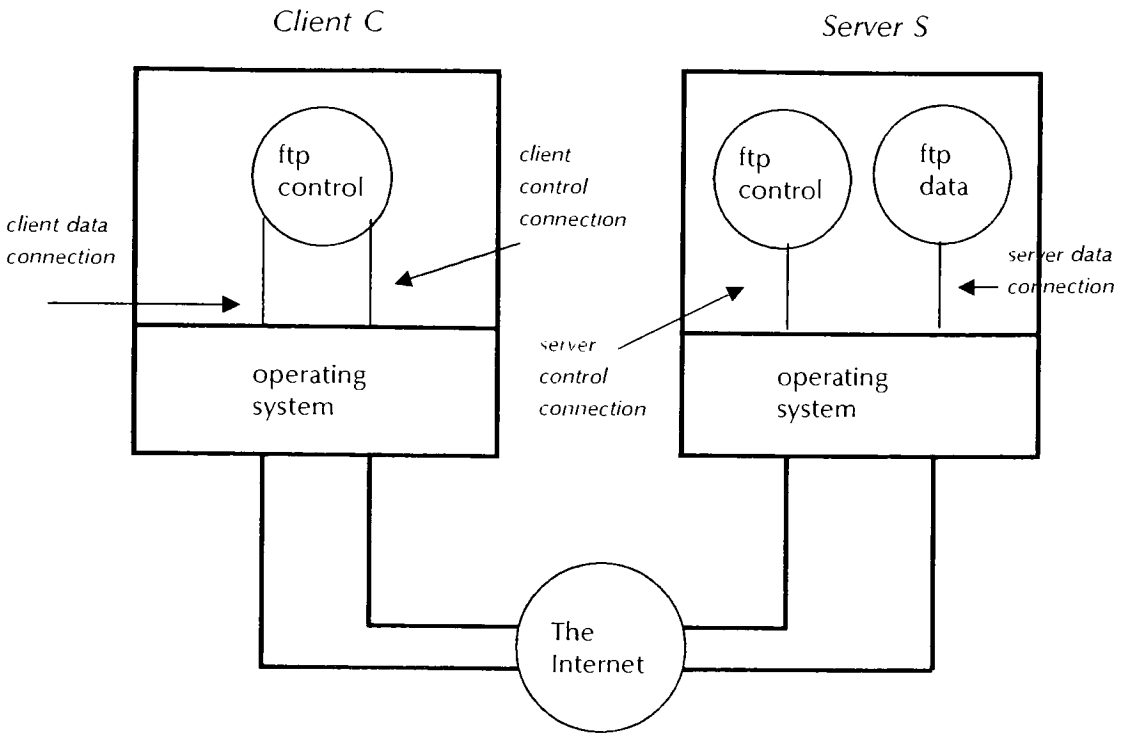


Figure 5. Simple FTP connection

connection. In fact, the protocol requires that the control connection be open while data transfer is in progress. *C* may then continue interaction with *S* or relinquish the control connection.

FTP has several provisions for handling file system differences between two end systems. It has an ASCII-EBCDIC conversion facility, it is able to translate the most common file formats, and it allows the user to specify how binary data is to be encoded between two end systems with varying word lengths. FTP also distinguishes between file-oriented transfers, record-oriented transfers, and page-oriented transfers. A file-oriented transfer is simply a continuous sequence of data bytes. In record-oriented transfers, the file is made up of sequential records. A random-access file is an example of a file where a page-oriented transfer should be specified. Although a file can be explicitly described using these parameters, the byte size used for transmission over the data connection is always 8 bits.

2.3 UNIX rcp

UNIX rcp (remote copy) is one of several remote execution commands first made available from the University of California at Berkeley's 4.x releases of the UNIX operating system. They are frequently referred to as the 4.xBSD "r" commands since they all begin with the letter r (rlogin, rsh, rcmd, rcp, etc.). Implementations of rcp are now available in many UNIX operating systems. Theoretically, rcp is a protocol that could be implemented on non-UNIX operating systems, but the author is unaware of any such implementations.

Berkeley has never published a formal specification of any of the protocols used for remote command execution. Consequently, it is not possible to describe rcp without referencing its implementation. This is in contrast to the other protocols in this study, which are defined by protocol specifications to which their implementations are required to adhere.

Remote command execution is when a process on a host computer causes a program to be executed on another host. Usually the client wants to pass data to the remote program, and capture its output also. What this means for UNIX is that the client needs to transmit data that becomes the standard input of the remote process and also needs to receive what the remote process writes to its standard output and standard error, as shown in Figure 6.

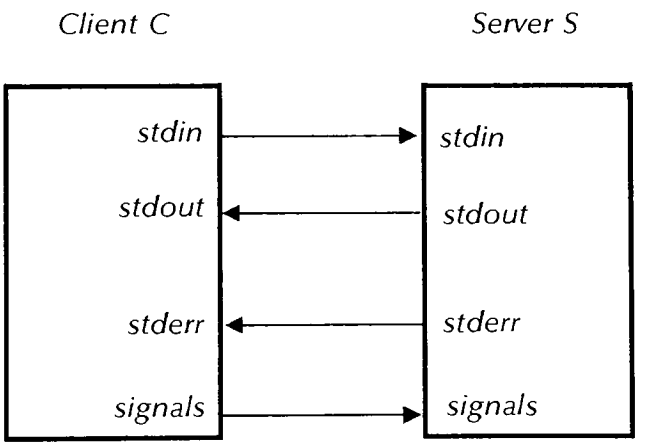


Figure 6. UNIX remote execution

UNIX rcp is used to copy files and directories between systems. It is a command/response protocol that behaves recursively when operating on directories. Like ftp, rcp is an application program that uses TCP at the transport level. Thus, both end systems are always in a mutually known state. The rcp protocol uses only one TCP connection between itself and the remote end. It does not use a second connection for stderr and signals. The rcp program functions as either client or server. When the remote copy of rcp is invoked, a flag indicates whether it is to send or receive a file.

The *rcmd* client function and its corresponding *rshd* server are central to the 4.xBSD networking system. UNIX rcp and many of the other "r" commands call the *rcmd* function. The *rshd* server is the server for both the *rcmd* function and the *rsh* program. The following is a model for a typical rcp copy:

Client:	Server:
the rcp client program is invoked, which in turn executes the *rcmd* client. rcmd creates a TCP socket with a reserved port.	
	the *rshd* server accepts the TCP connection.
rcmd sends a NULL byte, signaling that no secondary port for stderr and signals is required.	
	acknowledges receipt of the NULL string.
rcmd writes three ASCII strings to the server: client and server user name (usually the same), and the rcp command with the -f option, indicating it is to send a file.	
	reads the ASCII strings, validates the user and returns the validation.
rcmd receives the validation and returns a socket descriptor to the caller.	
	acknowledges the socket descriptor and invokes the shell to execute the rcp command for the client.

sends
"C mode filesize destfilename"

opens the file to be copied and sends a NULL byte. Sends the complete file followed by a NULL byte.

reccives the file and sends
"E" to end the session.

acknowledges the session end with a NULL byte.

Once preliminary negotiations are finished, the data that moves between the two end systems is typically in the form of one of the following six commands:

- C *mode filesize destinationfilename*
  This command copies a file. When the *destinationfilename* is successfully opened for writing, the client sends the source file to the server.

- D *mode filesize destinationfilename*
  This command copies a directory. Recursion is used to copy the entire contents of the directory. The command ends when the server receives the E command or a fatal error occurs.

- E
  This command signals the end of the session or the end of a directory.

- T t1 t2 t3 t4
  This command sets the modification and access times for the file(s) that follow. t1 represents modification time in seconds since 1/1/70. t2 represents modification time in microseconds. t3 represents access time in seconds since 1/1/70. t4 represents access time in microseconds.

- \01 *errormessagetext*
  This command signals a non-fatal error.

- \02 *errormessagetext*
  This command signals a fatal error.

2.4 XNS Filing

Xerox Network Systems (XNS) was born in the early 1980s when Xerox scientists began modifying PUP (PARC Universal Packet protocols) to create a more robust product. The XNS architecture defines a series of protocols for use between systems in a networked environment. One of the application-level protocols, the Filing Protocol, defines a general purpose file management system which is hierarchical in nature and supports a wide variety of functions.

The protocol defines the interaction between a filing client and a file service. The Filing Protocol follows a session-oriented model in which a client interacts on behalf of a human or non-human user. The session is established when the client successfully logs on to the service, and is terminated either at the request of the client, or at the discretion of the file service. The Filing Protocol provides a robust set of functions, including file transfer commands, session management commands, file access and management commands, and directory management commands.

The XNS remote procedure call protocol is called Courier [Xerox81b] and defines a request-reply discipline used by higher level application protocols, such as Filing. Courier takes a local function call and transfers it to a remote resource, such as a file server, for execution. A Courier call is analogous to a subroutine call where arguments are passed and values may be returned, as shown in Figure 7.

Large data items, such as directory listings and the contents of a file, are not easily modeled as procedure arguments and results. For this reason, Courier also includes a Bulk Data Protocol which defines the mechanism for transmitting simple streams of data between two Courier applications.

Courier's responsibility is to interface between applications such as XNS Filing and a transport. In a full XNS stack the transport is typically Sequenced Packet Protocol (SPP). Courier is internally divided into three hierarchical layers. At the top layer, a message stream carries the calls and replies between the systems. The middle layer is called the object stream, which carries structured data such as booleans and cardinals, in a machine independent way. This layer resolves differences in data byte ordering, data type size, representation, and alignment. The bottom stream layer carries blocks of data between the

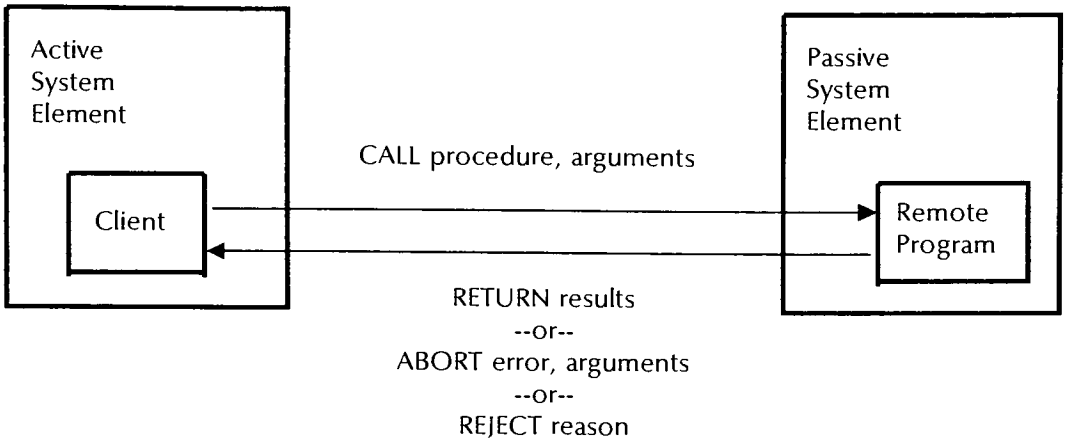


Figure 7. The Courier model

two systems. Courier includes not only a remote procedure call mechanism, but a data description language to describe its data.

Like FTAM, XNS Filing views a file as a body of data consisting of two types of information: content and attributes. The content of a file is the data actually contained within the file. Attributes are data items that identify the file and describe its contents. Attributes can vary widely in purpose, structure, and behavior. Just as a file's content can be modified so can its attributes. FileID, name and type are three examples of file attributes.

Before making use of a file service, a client must log on and present user credentials. The service responds by establishing a session and returning a session handle. The session handle identifies the client in future requests. The session keeps track of files that are open and any file locks. To manipulate a file, a client must open that file. When a new file is created or an existing file is opened, the file service returns a file handle. The file handle is presented in subsequent operations to identify this file to the file service. When interaction is complete, the client logs off.

At the transport level, the Sequenced Packet Protocol (SPP) provides for the reliable delivery of packets from source to destination. Like TCP, SPP guarantees a sequenced, flow controlled, reliable virtual circuit.

Until recently, XNS services were available primarily to users of Xerox workstations and other Xerox office automation equipment, and to users of large minicomputers. The widespread availability of relatively inexpensive personal workstations has made it feasible and desirable to extend XNS services to user of these UNIX workstations. XNS for UNIX V.3 is an implementation of XNS that operates on workstations and minicomputers that use the UNIX operating system. This implementation allows workstation users access to the sophisticated printing, filing, and data transfer capabilities of Xerox products that speak XNS.

2.5 NFS

The Sun Network File System (NFS) is a facility for sharing files in a network of machines. NFS brings some of the advantages of a timesharing environment to the workstation world by providing the illusion that disks from one computer are directly connected to other computers. NFS allows multiple machines to combine their file systems as if they were all on one large computer. Users can move around in the file system, reading and writing to files, without needing to know where the files actually reside.

NFS was originally designed for UNIX computers in a workgroup environment. It was developed in 1984 by modifying the Berkeley 4.2 UNIX kernel. The resulting operating system was named SunOs 3.0. Despite its strong UNIX influence, NFS has been successfully implemented on at least 25 different vendors' hardware and under at least six non-UNIX operating systems [RSand89].

The NFS model is quite straightforward for a UNIX end user. The client builds its view of the file system using the MOUNT command. Any file systems that have been exported by a server and mounted by the client are available to the client (subject to access control), creating the illusion that the file system is in the user's local space. Users are able to invoke the same commands to access remote files as they do for local files.

NFS is a stateless protocol, that is, the server does not remember anything about clients between transactions. A client does not open a session with the server nor open a file on the server. This greatly simplifies the protocol, makes it easier to implement, and simplifies crash recovery. The statelessness also means that NFS does not need to use the services of a confirmed service transport entity. Sun's implementation of NFS uses the User Datagram Protocol (UDP from the Internet protocol suite) at its transport level and Internet Protocol (IP) at its network level.

A traditional UNIX file system is composed of directories and files, each of which has a corresponding index node (i-node), containing administrative information about the file. I-nodes are assigned unique numbers within a file system, but a file on one file system could have the same i-node as a file on another file system. To solve this problem, Sun has designed the virtual file

system (VFS), based on the vnode, a generalized implementation of i-nodes that are unique across file systems. Figure 8 is a schematic diagram of a file system interface and how NFS uses it. If a UNIX system call is requested for a local file, VFS will direct it to proceed as in traditional UNIX. If it is for a remote file that has been locally mounted, VFS will direct NFS to utilize the Sun RPC facility to access the remote file. All this detail is completely hidden from the user.

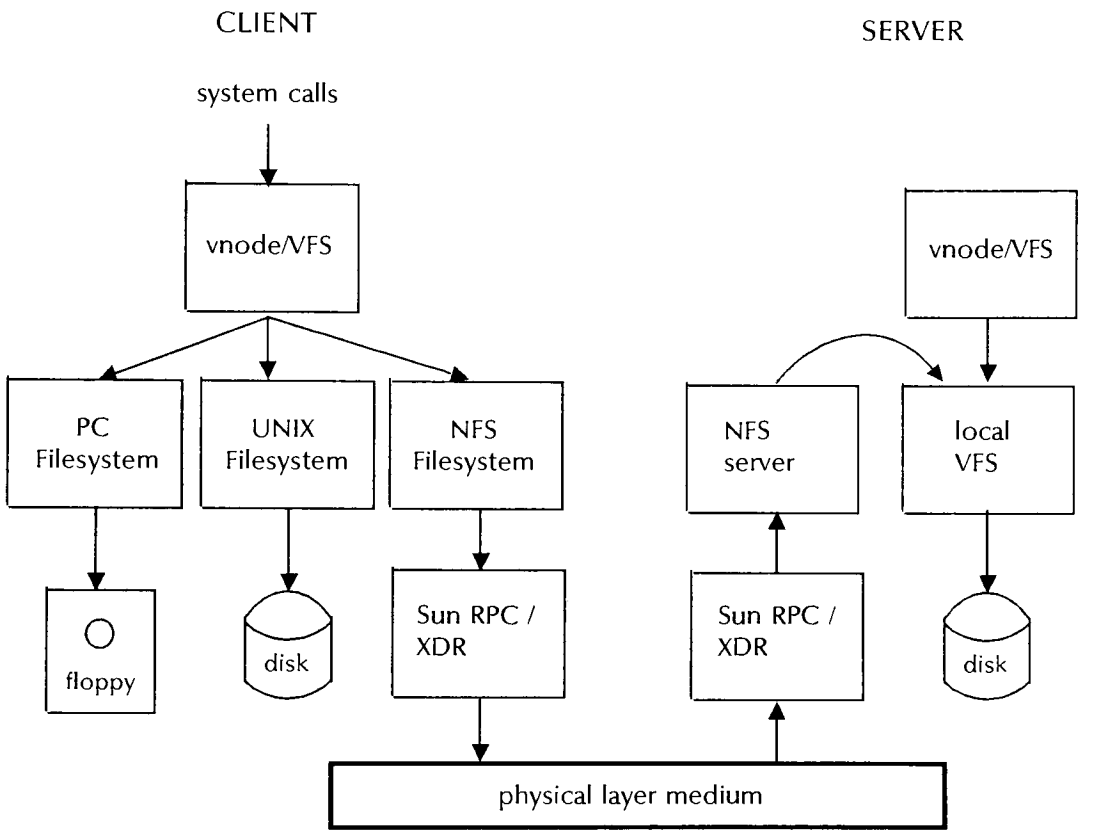


Figure 8. Flow of a NFS client's request to various file systems

A popular implementation of NFS from Sun Microsystems, Inc. is composed of a modified UNIX kernel, a set of library routines, and a collection of utility commands. The NFS protocol sits on top of two other protocols, Sun's Remote Procedure Call package (Sun RPC) and External Data Representation (XDR). XDR describes data in a machine independent way, allowing a variety of machines to communicate on the network.

NFS, Sun RPC, and XDR are de facto standards. No standards body has declared these protocols as official standards. However, an organization of over 16 vendors recently endorsed Sun RPC as a standard method of inter-machine communication [FGrec90].

XDR ensures a standard means of exchanging data across a network composed of heterogeneous machines. The XDR format removes machine dependencies such as byte ordering, floating point representation, and word lengths. This spares the application from having to know the original representation of the data elements. The data is converted from the local machine's representation to XDR format before being sent over the network. When it reaches the target host it is converted from XDR to the target machine's format. XDR defines a representation for the most commonly used primitive data types. In addition, complex data types can be constructed from the provided primitives. RPC handles setting up the XDR structure automatically.

Sun RPC uses the XDR specification to transfer its RPC header information between end systems. The user's data is not converted or manipulated in any way. It is presented to the client in a bit stream just as if it came off of the server's disk. In situations where the two end systems have different byte ordering schemes, a utility to convert the file into the proper byte order must be used by the client.

	FTAM	FTP	UNIX rcp	XNS Filing	NFS
formal protocol specification exists	Yes	Yes	No	Yes	Yes
filing protocol class	general-purpose file transfer system	general-purpose file transfer system	general-purpose file transfer system	general-purpose file transfer system	network file system
RPC-based	No	No	No	Yes, uses Courier	Yes, uses Sun RPC
machine-independent data representation	Yes, uses ASN.1 and BER	Yes, built into FTP	No	Yes, uses object layer of Courier	Yes, uses XDR
session-based protocol	Yes	Yes	No	Yes	No
preferred transport protocol	TP	TCP	TCP	SPP	UDP
type of transport	connection oriented	connection oriented	connection oriented	connection oriented	connectionless
number of network connections required by the protocol	1	2	1	1	0
separate mechanism for bulk data	Yes	Yes	No	Yes	No
files represented by attributes and content	Yes	No	No	Yes	Yes
third party transfers	No	Yes	Yes	Yes	No
approximate date of first implementation	late 1980s	late 1970s	mid 1980s	early 1980s	mid 1980s

Figure 9. Comparison of Protocols

3.0 Protocol Design Descriptions

Five key design aspects of filing protocols will be described in this section: exported interface, concurrency control, access control, error recovery, and performance. Details about the five protocol design choices follow. A protocol's *exported interface* is the set of operations available to an upper level application. The exported interface also defines the parameter data associated with each operation and the valid sequences of operations. *Concurrency control* addresses how the protocol ensures that the user has a consistent view of the file. One goal of concurrency control is to prevent two processes from modifying the same file at the same time. *Access controls* define users' access privileges to the use of a system, and to the files in that system. Access controls are necessary to prevent unauthorized or accidental use of files. The ultimate goal of *error recovery* is to provide uninterrupted service to users, despite the occurrence of various types of errors, such as failure of either host system or a transfer process. Each protocol's approach to handling such errors will be described.

Performance refers to the speed and efficiency of file transfers for a representative implementation. The file transfers measured in this study were of varying file sizes. The transfers were bidirectional between the two hosts, that is, the client both stored and retrieved files. This compares the symmetry of the protocol's implementation by measuring whether the transfer performance is better in one direction than the other.

3.1 Exported Interface

A protocol's exported interface is the set of operations available to an upper level application. It forms a boundary between the protocol itself and the applications that use the protocol. For remote procedure call-based protocols, the exported interface is the set of remote procedures and data parameters. For asynchronous message-based protocols, the exported interface is the set of commands and corresponding arguments. A table-formatted comparison chart summarizing the protocols' exported interface is included at the end of this section in Figure 10.

FTAM's exported interface:

Part of FTAM's specification is the ISO File Service Definition [ISO88c], which explicitly defines the exported interface. This definition describes 30 file service primitives and over 30 parameter data items. Operations may be performed on a file as a whole or on a portion of the file. The services provided to the FTAM users are modeled as service elements and their corresponding service primitives. The file service primitives are F-INITIALIZE, F-TERMINATE, F-U-ABORT, F-P-ABORT, F-SELECT, F-DESELECT, F-CREATE, F-DELETE, F-READ-ATTRIB, F-CHANGE-ATTRIB, F-OPEN, F-CLOSE, F-BEGIN-GROUP, F-END-GROUP, F-RECOVER, F-LOCATE, F-ERASE, F-CONNECT, F-DISCONNECT, F-READ, F-WRITE, F-DATA, F-DATA-END, F-TYPE-DEFINE, F-CANCEL, F-TRANSFER-END, F-CHECK, F-RESELECT, F-REOPEN, and F-RESTART.

From the total set of over 30 FTAM parameter data items, there are fifteen primary ones which are used in more than one file primitive:

access passwords	are the passwords associated with the actions specified in the *requested access* parameter.
account	identifies the account to which costs are to be charged.
action result	summarizes the *diagnostic* parameter.

activity identifier	an unambiguous value allocated to a file activity, used to re-establish the data transfer in case of errors.
attributes	are a list of the file's attribute names and values.
bulk data transfer specification	identifies the source or the target of the data transfer.
charging	contains cost information.
checkpoint identifier	an unambiguous integer which allows reference to the checkpoints.
concurrency control	specifies which locks are required--Not Required, Shared, Exclusive, or No Access.
diagnostic	conveys detailed information on the failure of a requested action.
FADU identity	specifies the target FADU to which a series of one or more operations is related.
FADU lock	sets individual FADU locks on or off.
requested access	gives the actions to be performed on the file while selected.
shared ASE Information	allows information of other Application Service Entities to be associated with FTAM primitives.
state result	indicates a success or failure. This parameter is used on primitives which force state changes, such as F-DESELECT.

It is not necessary for a FTAM implementation to define all the file service primitives and handle all the parameter data items. Instead, it can implement one of five service subsets: kernel, simple management, management, access, and error. The kernel subset constitutes the minimum requirements of all

implementations and includes what is needed for a file transfer without error recovery. The primitives of the kernel subset are F-CONNECT, F-DISCONNECT, F-P-ABORT, F-U-ABORT, F-SELECT, F-DESELECT, F-OPEN, F-CLOSE, and at least one of the following: (F-READ and F-WRITE), (F-DATA and F-DATA-END), or F-CANCEL. The other four subsets are built by adding commands and additional functionality to the kernel service subset.

FTP's exported interface:

The 33 commands that make up FTP's exported interface can be partitioned into three categories: those commands specifying access control identifiers, those commands defining data transfer parameters, and FTP service requests. The eight access control commands are USER NAME, PASSWORD, ACCOUNT, CHANGE WORKING DIRECTORY, CHANGE TO PARENT DIRECTORY, STRUCTURE MOUNT, REINITIALIZE, AND LOGOUT.

All data transfer parameters have default values, and the commands that specify data transfer parameters are required only if the default values are to be changed. The default is either the last specified value or the standard default value. The five transfer parameters and their standard default values are DATA PORT, PASSIVE, REPRESENTATION TYPE (Ascii non-print), FILE STRUCTURE (File), and TRANSFER MODE (Stream).

The FTP service commands define the file transfer requested by the user. The argument of these commands will normally be a file pathname, which adheres to the conventions of the server site. Below are the twenty FTP service requests: RETRIEVE, STORE, STORE UNIQUE, APPEND, ALLOCATE, RESTART, RENAME FROM, RENAME TO, ABORT, DELETE, REMOVE DIRECTORY, MAKE DIRECTORY, PRINT WORKING DIRECTORY, LIST, NAME LIST, SITE PARAMETERS, SYSTEM, STATUS, HELP, and NOOP.

FTP has twelve command arguments:

account-information	identifies the user's account.
byte size	decimal integer to indicate the byte size.
form-code	N-Nonprint, T-Telnet, or C-Carriage Control.

host-port	the concatenation of the internet host address and the TCP port address.
marker	data checkpoint where file transfer is to be restarted.
mode-code	S-Stream, B-Block, or C-Compressed.
password	identifies the user's password.
pathname	specifies a directory or other system-dependent file group designator.
string	either a system name for the SITE command or a command name for the HELP command.
structure-code	F-File, R-Record, or P-Page.
type-code	A-ASCII, E-EBCDIC, I-Image, or L-Local byte.
username	is the required identification for access to a file system.

Like FTAM, the FTP protocol specifies a minimum implementation of nine commands: USER NAME, QUIT, DATA PORT, REPRESENTATION TYPE, MODE, STRUCTURE MOUNT, RETRIEVE, STORE, and NOOP.

UNIX rcp's exported interface:

Since UNIX rcp can only be invoked from the UNIX command prompt, its exported interface is the command line itself, which has the following format:

```
rcp [-p] filename1 filename2
      or
rcp [-pr] filename directory
```

The -p option attempts to give each copy the same modification times, access times, and modes as the original file. The -r option copies each subtree rooted at filename. The filename or directory can be in one of the following forms:

hostname:path
username@hostname:path
username@host.domain:path
local filename containing no colons

If a full pathname is not specified, it is interpreted relative to the home directory.

XNS Filing's exported interface:

The XNS Filing exported interface is a collection of 23 remote procedures and fourteen parameter data items. LOGON, LOGOFF, and CONTINUE are used to initiate, terminate, or continue a session. Files may then be OPENed or CLOSEd. Retrieving or modifying a file's permissions are done with GETCONTROLS and CHANGECONTROLS. A file's attributes are manipulated with GETATTRIBUTES, CHANGEATTRIBUTES, and UNIFYACCESSLISTS. Files can be located (FIND), LISTEd, STOREd, RETRIEVEd, REPLACEd, CREATEd, DELETEd, COPYed, and MOVEd. Partial file contents are manipulated with RETRIEVEBYTES and REPLACEBYTES. SERIALIZE operates on a subtree of files by encapsulating the file's content, attributes, and descendants. This encapsulation is often a useful entity to work with when transferring a directory to another file service. DESERIALIZE reconstructs the file's content, attributes and descendants.

XNS has fourteen data types used as parameter data in the server procedures:

AttributeSequence — is a collection of file attributes that identify the file and describe its contents. Examples of file attributes are fileID and name.

AttributeTypeSequence — is a collection of file attribute types.

BulkData.Sink — is an address to receive the requested data.

BulkData.Source — is the address to supply data.

ByteRange — specifies a contiguous sequence of bytes within the content of a file.

Cardinal	is a number in this case used by the Continue procedure to specify seconds.
Clearinghouse.Name	is the distinguished name of the file service to be accessed by the session. (Used only by the Logon procedure.)
ControlSequence	is a set of controls which characterize the intended use of a file handle.
ControlSequenceTypes	is a sequence of the types of file handle control items. There are 3 control types: lock, timeout, and access.
Credentials	represent the filing client's proof of identity. (Used only by the Logon procedure.)
Handle	is a unique way to identify a file within a client's filing session.
ScopeSequence	is a sequence of characteristics that describe the files of interest, and how they are to be examined. Examples of scopes are count and direction. ScopeSequences are used only by the Find and List procedures.
Session	encapsulates the state of the client. It keeps track of open files, locks, and the client username. This data parameter is used by all 23 procedures.
Verifier	substantiates that all procedure calls using the session handle originated from the same client. (Used only by the Logon procedure.)

Like FTAM and FTP, XNS defines a minimal capability to store, retrieve, enumerate, and delete files of a remote service. This minimum implementation is called the Filing Subset. The subset is made up of nine procedures: LOGON, LOGOFF, CONTINUE, OPEN, CLOSE, RETRIEVE, STORE, LIST, and DELETE.

NFS's exported interface:

NFS's exported interface is its seventeen server remote procedures, most with self-explanatory procedure names: Get File Attributes, Set File Attributes, Look Up File Name, Read from Symbolic Link, Read from File, Write to Cache, Write to File, Create File, Remove File, Rename File, Create Link to File, Create Symbolic Link, Create Directory, Remove Directory, Read from Directory, Get Filesystem Attributes, and Do Nothing.

All of the procedures in the NFS protocol are assumed to be synchronous. When a procedure returns to the client, the client can assume that the operation has completed and any data associated with the request is on stable storage. For example, when a WRITE request is returned to the client, it can assume that the write is safe, even in case of a server crash, and it can discard the data written. This is a very important part of the statelessness of the server.

NFS has eleven basic data types used as parameter data in the server procedures:

attrstat is a common procedure result indicating the status of the call and the file attributes.

diropargs is a structure used to identify directories.

diropres is a common procedure result for directory operations, indicating the status of the call and the file's handle and attributes.

fattr is a structure containing the fourteen attributes of a file.

fhandle is the most common NFS procedure parameter. It contains all the information needed to distinguish an individual file.

filename is the name of the file.

ftype gives the type of the file: nonfile, regular, directory, block-special device, character-special device, symbolic link.

path is the file pathname.

sattr contains the six file attributes which can be set from the client.

stat is returned with every procedure's results. Its value is either NFS_OK or one of seventeen error conditions.

timeval is a structure used to pass time and date information.

	FTAM	FTP	UNIX rcp	XNS Filing	NFS
exported interface	30 primitives and > 30 parameters	33 commands and 12 arguments	the rcp command itself	23 remote procedures and 14 parameter data types	17 remote procedures and 11 parameter data types
stated minimal implementation	Yes	Yes	No	Yes	No
session initialization	F-INITIALIZE, F-CONNECT	USER NAME, PASSWORD, ACCOUNT, REINITIALIZE	rcp [-p] filename1 filename2	LOGON	
session closure	F-TERMINATE, F-DISCONNECT	LOGOUT		LOGOFF	
session continuation				CONTINUE	
file open operation	F-OPEN			OPEN	
file close operation	F-CLOSE			CLOSE	
file selection	F-SELECT, F-DESELECT				
file creation	F-CREATE			CREATE	Create File

Figure 10. Comparison of Exported Interfaces

	FTAM	FTP	UNIX rcp	XNS Filing	NFS
file deletion	F-DELETE, F-ERASE	DELETE		DELETE	Remove File
file rename operation		RENAME FROM, RENAME TO		MOVE	Rename File
read operation	F-READ				Read from File, Read from Symbolic Link
write operation	F-WRITE			REPLACE	Write to Cache, Write to File
file retrieval		RETRIEVE		RETRIEVE	
file storing		STORE, STORE UNIQUE		STORE	
file location	F-LOCATE			FIND, LIST	Look Up File Name
partial file contents manipulation		APPEND		RETRIEVEBYTES, REPLACEBYTES	
file size allocation		ALLOCATE			
file attribute manipulation	F-READ-ATTRIB, F-CHANGE-ATTRIB			GETATTRIBUTES, CHANGEATTRIBUTES, UNIFYACCESSLISTS, GETCONTROLS, CHANGECONTROLS	Get File Attributes, Set File Attributes, Get Filesystem Attributes

Figure 10. Comparison of Exported Interfaces

	FTAM	FTP	UNIX rcp	XNS Filing	NFS
bulk data operations	F-DATA, F-DATA-END, F-CANCEL, F-TRANSFER-END			uses separate Bulk Data protocol	
file encapsulation				SERIALIZE, DESERIALIZE	
directory creation		MAKE DIRECTORY			Create Directory
directory deletion		REMOVE DIRECTORY			Remove Directory
other filesystem management		CHANGE WORKING DIRECTORY, CHANGE TO PARENT DIRECTORY, PRINT WORKING DIRECTORY, STRUCTURE MOUNT, LIST, NAME LIST		COPY	Read from Directory, Create Link to File, Create Symbolic Link
command abortion	F-U-ABORT, F-P-ABORT	ABORT			
null operation		NOOP			Do Nothing
filing operation grouping	F-BEGIN-GROUP, F-END-GROUP				

Figure 10. Comparison of Exported Interfaces

	FTAM	FTP	UNIX rcp	XNS Filing	NFS
error recovery	F-RECOVER, F-RESTART, F-CHECK, F-RESELECT, F-REOPEN	RESTART			
file descriptions	F-TYPE-DEFINE	REPRESENTATION TYPE, FILE STRUCTURE			
transfer mode specification		TRANSFER MODE			
help		HELP			
port manipulation		DATA PORT, PASSIVE			
services query		SITE PARAMETERS			
operating system query		SYSTEM			
file operation status query		STATUS			

Figure 10. Comparison of Exported Interfaces

3.2 Concurrency Control

The objective of a concurrency control mechanism is to ensure that a client has a consistent view of a file by restricting shared access. This mechanism is necessary to keep files that need to be simultaneously accessed by many users in a consistent and known state. It is undesirable to allow a client to read a file while another client is writing to the file. Similarly, two clients must not be allowed to concurrently write to identical parts of the same file. Protocol concurrency control mechanisms are designed to provide a way for a user to perform a coordinated series of actions without interference from concurrent accesses. A table-formatted comparison chart summarizing the protocols' concurrency control is included at the end of this section in Figure 11.

FTAM's concurrency control:

Many of FTAM's file service primitives make use of a concurrency control parameter, first described in Section 3.1. This indicates the relation of the current regime to other activities on the same file. A lock is the mechanism used to define the access available to other users. Four locks are available:

- *Not required*, which means the user requesting the lock will not perform the operation, but other users may.
- *Shared*, which means the user requesting the lock may perform the operation, and so may other users.
- *Exclusive*, which means the user requesting the lock may perform the operation and other users may not.
- *No access*, which means no user may perform the operation.

The locks may be placed on the following eight operations: read, insert, replace, extend, erase, read attribute, change attribute, and delete file.

These locks may be applied at two different levels: the outer level, which controls access to the whole file, or the inner level, which controls access to individual file access data units (FADUs). Outer level concurrency controls are applied at the time the file is selected or opened, and persist until the file is deselected or closed. For example, an application may specify shared read

access and no access to all other actions. Any number of users can read data, but if a user requests an exclusive replace lock, it will not be granted until all other access has ceased, thereby maintaining overall data integrity.

To allow a finer level of granularity, FADU locking can be requested. The concurrency controls specified at the time the file is opened are not applied immediately. Instead, the control requested is applied for the duration of each file access action. In addition, FADUs can have a lock applied for some period within the file open regime, bounded by a pair of access actions marked in the protocol.

FTP's concurrency control:

The FTP protocol specification does not describe how to mark, detect, or unmark a "busy" file. The only concurrency control provisions found in the specification is a brief description of reply code #450, "Requested file action not taken. File unavailable (e.g., file busy)" [DDN85].

At a minimum, most implementations will rely on its local operating system's concurrency control mechanisms, although this is not an explicit requirement of the protocol. This is true for the UNIX implementation used in this study.

UNIX rcp's concurrency control:

The UNIX rcp protocol has no explicit concurrency control capabilities. It relies on the UNIX operating system to control simultaneous access to files. That is, when a user has a file open for writing, a second user should not be allowed to make a remote copy of that file. Likewise, while a file is being copied by rcp, other users should be allowed to open that file for reading, but not for writing.

XNS Filing's concurrency control:

There are two levels of concurrency control in XNS Filing, an implicit and an explicit level. The client never needs to explicitly acquire locks unless it wants additional protection over and above what the server implicitly provides.

When a file is opened with no explicit lock instructions from the client, the server ensures that no other session will move or delete the file while the client has it open. As the client executes procedures, the server automatically adds and removes locks based on the requirements of the called procedures. Whenever the contents of a file is read, a share lock is placed on the file. Whenever a client changes the content or attributes of a file, or when children are added to or removed from a directory, an exclusive lock is placed on the file. A *share* lock will prevent other sessions from acquiring an exclusive lock on the file. An *exclusive* lock will prevent other sessions from acquiring a share or an exclusive lock on the file. If extra protection is required, the client can place a share or an exclusive lock on the file at the time it is opened.

When a client requests a lock that is unavailable, the file service waits until it becomes available or until a predetermined timeout expires, whichever occurs first. The length of the timeout period is an implementation-dependent constant, but may be modified by the client.

NFS's concurrency control:

NFS is a stateless protocol, but the concept of concurrency control is inherently stateful. Being stateless, NFS does not support remote file locking. Instead, there is a separate, remote procedure call-based Network Lock Manager to handle concurrency control. The Lock Manager follows the industry standard for file and record locking as defined by the AT&T System V Interface Definition (SVID) [Sun90]. In the UNIX implementation, file modifications are locked at the i-node level. Because the NFS server maintains no locks between requests and a WRITE may span several Sun RPC requests, two clients writing to the same remote file CAN receive intermixed data on long writes [RSand89].

	FTAM	FTP	UNIX rcp	XNS Filing	NFS
concurrency control mechanisms	uses 4 file locks: not required, shared, exclusive, and no access	implementation dependent	implementation dependent	uses 2 file locks: share and exclusive	implementation dependent
file-level locks	Yes	not applicable	not applicable	Yes	not applicable
record-level locks	Yes	not applicable	not applicable	No	not applicable

Figure 11. Comparison of Concurrency Control

3.3 Access Control

Access control services are not unlike the services provided by a security guard. Access control mechanisms police the accession of files and only permit access to authorized users. The lack of access control mechanisms in a protocol may result in unauthorized or accidental use of the files. A related access control issue is how each protocol handles sensitive password data. A table-formatted comparison chart summarizing the protocols' access control is included at the end of this section in Figure 12.

FTAM's access control:

FTAM's access control mechanisms are based on the concept of an access control list (ACL). The ACL is a permanent property of the file, and is stored for as long as the file exists. An access check is made against this list upon each user request. FTAM sends the ID and the password of the initiator to the remote system each time either side wants to perform a file operation. Each entry in the ACL gives a set of actions and a set of tests which an initiator needs to satisfy before the filestore operations can be performed. For example, an ACL might contain an entry allowing a number of named initiators to read a file, and a separate entry allowing any entity supplying a particular password to write to it.

FTAM maintains a collection of access passwords for every file. The implementation makes the decision how to store these passwords--either as an unencrypted character field or an encrypted "octet string", utilizing a public key encryption scheme.

FTP's access control:

The FTP protocol specification is short on detail regarding access control issues. "It is the prerogative of a server-FTP process to invoke access controls" [DDN85]. Reply code #550 is used to indicate that a "requested action was not taken, as the file is unavailable (for example, file not found, no access)". Most implementations will at least implement its local operating system's

access control restrictions. This is true for the UNIX implementation used in this study.

The protocol specification recognizes that password data is quite sensitive, and places the responsibility on the client to hide password information when it is sent across the network. This is because the server has no "foolproof way to achieve this." Suggestions from the specification include suppressing typeout and "masking" the password [DDN85]. In many FTP implementations, including the one used in this study, the password is sent across the network unencrypted.

UNIX rcp's access control:

The network security features provided by 4.xBSD were designed to operate in an open environment where some hosts were designated as "trusted". The rshd server bases its authentication on the Internet address of the client--both the 32-bit Internet network ID and host ID, and the 16-bit TCP port number.

rshd requires two login names to accompany each request for service: the name of the user on the client's system making the request, and the name of the user on the server's system. rcp typically invokes rshd with an identical client and server user name.

A series of three security checks is then performed by rshd. All three checks must pass before connection to the remote host is granted:

1) The client's host address is verified calling the *gethostbyaddr* and *gethostbyname* library functions.

2) The password entry file on the server is checked for the server's user name.

3) The */etc/hosts.equiv* file on the server's machine is checked for the client's system name. This file contains those hosts considered "trusted" by the server. If found, authentication is considered successful. Otherwise, the *.rhosts* file in the home directory of the server's user name is checked for the client's system name and

client user name. If found, authentication is considered successful. If not, connection to the remote host is denied.

In UNIX rcp, password data is never transmitted across the network, thereby eliminating the need for an encryption strategy.

XNS Filing's access control:

An XNS filing client may specify two types of controls for a file: who may access the file and what file operations are allowed by that user. This is implemented using the *accessList* file attribute. The *accessList* specifies the permissions to be granted to particular clients. Each enabled permission permits particular types of access to the specified client. Clients not appearing in the access list of a file are not allowed any access to the file.

There are five possible general operations on an XNS file: read, write, add, remove, and owner operations.

Read allows the client to read the file's content and attributes. For directory files, the client may list its children and search for files in the directory.

Write allows the client to change the file's content and attributes and to delete the file. For directory files, the client may change environment attributes and access lists of the directory's children.

Add is a permission reserved only for directory files. This permission allows the client to add children to the directory.

Remove is a permission reserved only for directory files. This permission allows the client to remove children from the directory.

Owner operations allow the client to change the file's access list.

If a permission for a particular operation has not been enabled, the server rejects any such requests for that file handle. There is special permission called *fullAccess*, which denotes permission to read, write, add, remove, and perform owner operations on files.

XNS Filing's user passwords are never transmitted across the network unencrypted. The password is used in conjunction with the Authentication Protocol [Xerox86], where it is immediately encrypted using a hashing algorithm or a NBS Data Encryption algorithm.

NFS's access control:

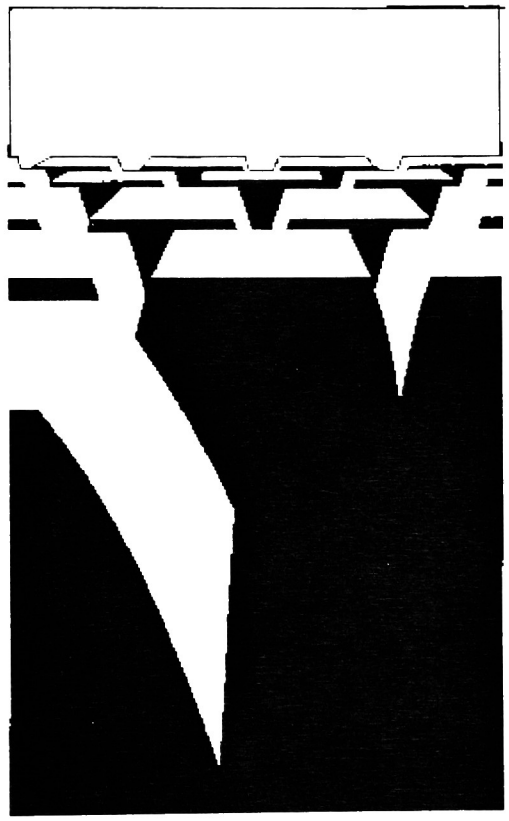
The NFS protocol does not explicitly define the permission checking used by servers. The server can elect to utilize the operating system's permission checking. Sun RPC includes a slot for authentication parameters on every call. The contents of the authentication parameters are determined by the type of authentication used by the server and the client. For example, no authentication can be used (AUTH_NONE) or UNIX-style permission checking (AUTH_UNIX). Two errors signal when access control is violated. The NFSERR_PERM error is invoked when the caller does not have the correct ownership to perform the requested operation. The NFSERR_ACCES error is invoked when the caller does not have the correct permission to perform the requested operation.

When UNIX-style permission checking is implemented, NFS makes use of UNIX's underlying file protection mechanisms. Each Sun RPC request from a client carries the identity of the requester. The server temporarily assumes this identity, and file permissions are checked exactly as if the user had logged in directly to the server.

All NFS file data, including password data, is transmitted across the network in RPC packets. These packets are not encrypted, increasing the importance of a physically secure network.

	FTAM	FTP	UNIX rcp	XNS Filing	NFS
access control	uses ACL mechanism	implementation dependent	implementation dependent	uses ACL mechanism	implementation dependent
password handling	implementation dependent	implementation dependent	not encrypted	always encrypted	not encrypted

Figure 12. Comparison of Access Control



3.4 Error Recovery

The ultimate goal of error recovery is to provide uninterrupted service to users. This section discusses each protocol's handling of gross errors, such as network and host failures. Some protocols have a separate application-level error recovery scheme, while others rely on the underlying protocols. Typically, a filing protocol will rely on its underlying transport protocol to detect and correct smaller-scale errors, such as lost or scrambled bits. A table-formatted comparison chart summarizing the protocols' error recovery is included at the end of this section in Figure 13.

FTAM's error recovery:

FTAM provides two styles of error recovery, both of which depend on the marking of checkpoints within the data transferred. One, called Restart, allows for resynchronization to a checkpoint following the discovery of an error which did not interrupt the supporting communication. The other, called Recover, allows for an activity to be resumed following a more complete failure.

The high-level association between the communicating applications is maintained when both systems hold matching records of the transfer, called dockets. Checkpoints are inserted into the stream of data at points chosen by the sender. Both the sender and receiver save the position in the file of any active checkpoints in the dockets. The type of non-volatile memory storage used to hold the docket is implementation-dependent. The error recovery protocol reconstructs the supporting connections and the state of the data transfer prior to the failure, based on the information held in the dockets, and the communication can continue.

The choice of which error recovery mechanism to use or whether to rely on the inherent mechanisms in the supporting layers will depend on the reliability objectives of the file service. To date, one of the few companies to implement FTAM's restart and recovery functional units is Tecsiel of Pisa, Italy [LMant89].

FTP's error recovery:

For block and compressed modes of data transfer, the FTP protocol includes a RESTART procedure to protect its users from system errors. It requires the sender of data to insert a special marker code in the data stream with some marker information. The marker could represent a bit-count, a record-count, or any other information by which a system may identify a data checkpoint. The receiver of data would then mark the corresponding position of this marker in the receiving system, and convey the marker information to the user. When a system fails, the user can issue a RESTART command with the server's marker code as its argument. It should be immediately followed by the command which was being executed when the system failure occurred. This causes the sender to position itself in the file at the specified marker point and re-send the file from that starting point. The receiver is responsible for joining the incoming data to the first part of the file. This joining process becomes complex when the hosts use different character sets or file structures.

Those implementations of FTP that do not support block and compressed modes of data transfer do not have any error recovery capabilities. The UNIX implementation used in this study does not have any error recovery capabilities since it only supports the stream mode of data transfer.

UNIX rcp's error recovery:

The rcp protocol does not have any facilities for recovering from system or network failures.

XNS Filing's error recovery:

In XNS, when an abnormal condition arises during execution of a remote procedure, the file service makes every effort to undo the effects of the partial execution so that the file service appears to the client as though the procedure had never been called. The file service does not guarantee that such effects can always be reversed.

NFS's error recovery:

The NFS interface is defined so a server can be stateless, an advantage in the event of a crash. This means that a server does not have to remember from one transaction to the next anything about its clients, transactions completed, or files operated on. Client workstations can continue to operate even when the server crashes and reboots. When an NFS server crashes or a packet gets lost between the server and the client, the client will continue to send requests until it gets an answer. These retransmissions are performed by Sun RPC. The client should not be able to tell the difference between a server that has crashed and recovered, and a server that is slow.

	FTAM	FTP	UNIX rcp	XNS Filing	NFS
error recovery	uses checkpoint and docket mechanism	none for stream mode; uses checkpoint mechanism for block and compressed mode	none	remote procedure attempts to undo its operations	none required because of stateless server model

Figure 13. Comparison of Error Recovery

3.5 Performance

It is difficult to compare protocols abstractly without referencing a specific environment. The primary focus of this paper is the comparison of protocols, not the comparison of implementations. Often, this separation is difficult to maintain. A key component of the comparison is performance, which can be measured only with a representative implementation. The UNIX-based reference implementations selected for this study are:

1) FTAM from the ISO Development Environment (ISODE) Version 6.0 for SunOS UNIX.

2) FTP from SunOS, Version 4.1.1.

3) UNIX rcp from SunOS, Version 4.1.1.

4) XNS Filing from xnsftp Revision 2.16 from XNS for UNIX.

5) NFS from SunOS, Version 4.1.1.

Figure 14 illustrates the chosen implementations' full 7-layer stack relative to the OSI networking model. A table-formatted comparison chart summarizing the protocols' performance is included at the end of this section in Figure 15.

As discussed in the Introduction, a somewhat equal performance comparison is possible only if these implementations use a common protocol at the transport level (layer 4). The implementations of FTAM, FTP, and UNIX rcp that were selected all use a TCP transport. Standard XNS Filing uses a SPP transport and standard NFS uses a UDP transport. The TCP port was successful only for XNS Filing. A previous port of XNS Printing to TCP proved to be a useful resource.

Four major changes were required to port XNS Filing to TCP. First, all calls to the Clearinghouse service were replaced with calls to the *gethostbyname* library function. Second, whenever an XNS-formatted address was composed, this was replaced with the composition of an Internet-formatted address. Third, all SPP socket calls were replaced with TCP socket calls. The chosen implementation of XNS Filing uses a SPP-based Authentication service. Rather than port this server to TCP, the last major change was to comment out all

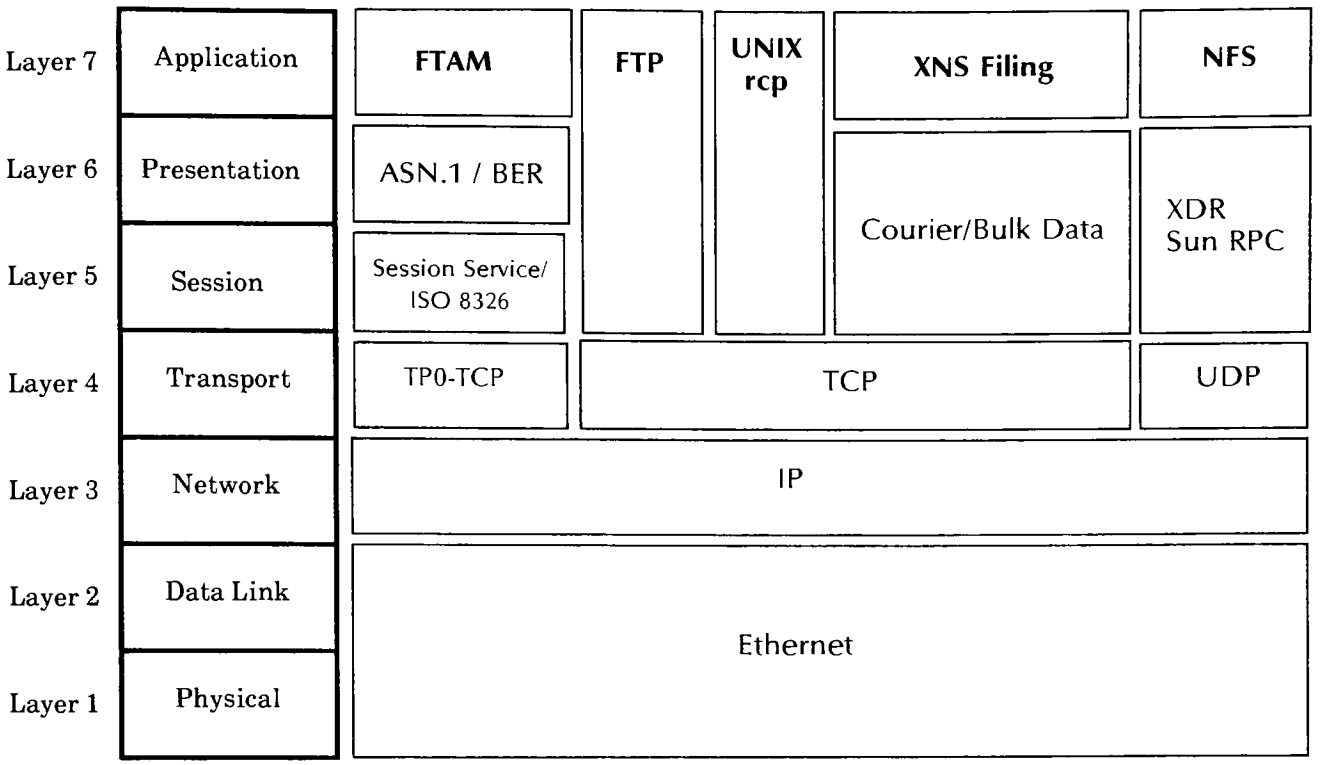


Figure 14. The chosen implementations' full 7-layer stack

calls to the Authentication service. Source code segments of these changes appear in Appendix C.

NFS, on the other hand, proved to be much more difficult. The port to TCP is inherently more difficult, as our only available NFS implementation uses UDP, a connectionless protocol, and the implementation resides in the UNIX kernel rather than in user space. This is in contrast to the implementations of FTAM, FTP, UNIX rcp, and XNS Filing, which are implemented outside the operating system kernel. The Reno release of Berkeley UNIX contains a non-proprietary implementation of NFS that can run over TCP, but this operating system does not run on Sun workstations. The introduction of a new hardware platform was undesirable, as consistency could no longer be claimed. The University of Michigan reportedly has a user space NFS server, but no client counterpart. Further details about other alternatives are not relevant here, but the result is that a TCP-based NFS was not used. This is mentioned here because it is only relevant to the performance comparison and not the other four comparison

areas. The reader should remember this when reading the performance statistics.

In order to analyze performance as a function of file size, 11 different files of varying sizes were transferred: 1 byte, 100,000 bytes, 200,000 bytes, 300,000 bytes, 400,000 bytes, 500,000 bytes, 600,000 bytes, 700,000 bytes, 800,000 bytes, 900,000 bytes, and 1,000,000 bytes. For each file size, the transfer is performed twenty times--ten times where the client is reading (pulling) the file and ten times where the client is writing to (pushing) the file. This push/pull sequence compares the symmetry of the protocol. For each set of ten transfers, the fastest and slowest transfers were discarded, and the remaining eight were averaged. To eliminate performance gains attributed to disk caching, ten uniquely named copies of the file were made before each exchange. The measurements made were using the binary transfer mode, which usually provides maximum performance because the end systems do not have to interpret each byte in the file. The UNIX implementations of FTAM, FTP and XNS require an explicit binary transfer mode request, where in NFS and UNIX rcp binary is the default mode.

First, a new version of each protocol implementation was built. The only difference in the new version and the standard implementation was the addition of timing procedures at the beginning and the end of the session. Special care was taken to time the entire session, for example from logon time to logoff time, rather than just the time required for file transfer. Otherwise, only the transport would be timed. The timing procedures call the *gettimeofday* library function. Second, UNIX shell scripts were written to automate the file transfer. The redirection of stdin was critical in this step in order to eliminates delays associated with data entry from the user. Otherwise there is potential for skewed measurements if the timer records the time it takes for the user to respond to a command prompt. Additional levels of shell scripts were written to perform the base script twenty times per protocol for eleven file sizes. The results were sent to a file, which then served as a data input file to SAS, a data analysis software system. SAS was programmed to discard the fastest and slowest transfers, average the remaining eight transfers and reorganize the results for an easy comparison among the five protocols. The source code for the timing routines, UNIX shell scripts, and SAS procedures appear in Appendix B.

The files were exchanged between a Sun Microsystems 4/260 workstation and a Sun Microsystems 4/330 workstation. The 4/260 was running SunOS Version 4.1 and the 4/330 was running SunOS 4.1.1. They were located on the same Ethernet local area network. All the measurements were made after hours on weekends, assuring that the Ethernet carried a minimal load. These tests will most likely yield different results when measured on other type of computers.

Figure 15 is a comparison plot of the implementations' performance while the client is writing to the server. Overall, FTP is the best performer across all file sizes. NFS's performance deteriorates dramatically as the file size increases. This is because a NFS write operation has to complete physically before returning to the client. Disk read operations, on the other hand, merely read from the standard UNIX disk cache. Consequently, NFS reads are fairly fast, and NFS writes can be slow. Several vendors are attempting to solve the problem with hardware so the NFS disk write can complete logically and return to the client. The data is stored in a battery backed RAM cache until a convenient time when it can be written to disk.

Figure 16 contains the same comparison as Figure 15 except it measures client read operations instead of writes. Here the clear winner is NFS. Its closest competitor is FTP, followed by UNIX rcp and FTAM. XNS Filing has the poorest performance here, requiring an average of 9.68 seconds to read 1 million bytes. The 1 million byte file transfer takes XNS Filing 79% longer than NFS, or 7.7 seconds longer.

Figures 17-21 take an individual look at each implementation, comparing client reads versus client writes. These plots present graphical descriptions of the protocol's symmetry. The most symmetrical protocols are FTAM and XNS Filing, as shown in Figure 17 and Figure 20. In contrast, NFS's strong asymmetry is illustrated in Figure 21, taking almost eight times as long to write a 1 million byte file as it does to read it. As filesize increases, FTP's symmetry weakens.

Performance of Filing Protocols
Client Writing to Server

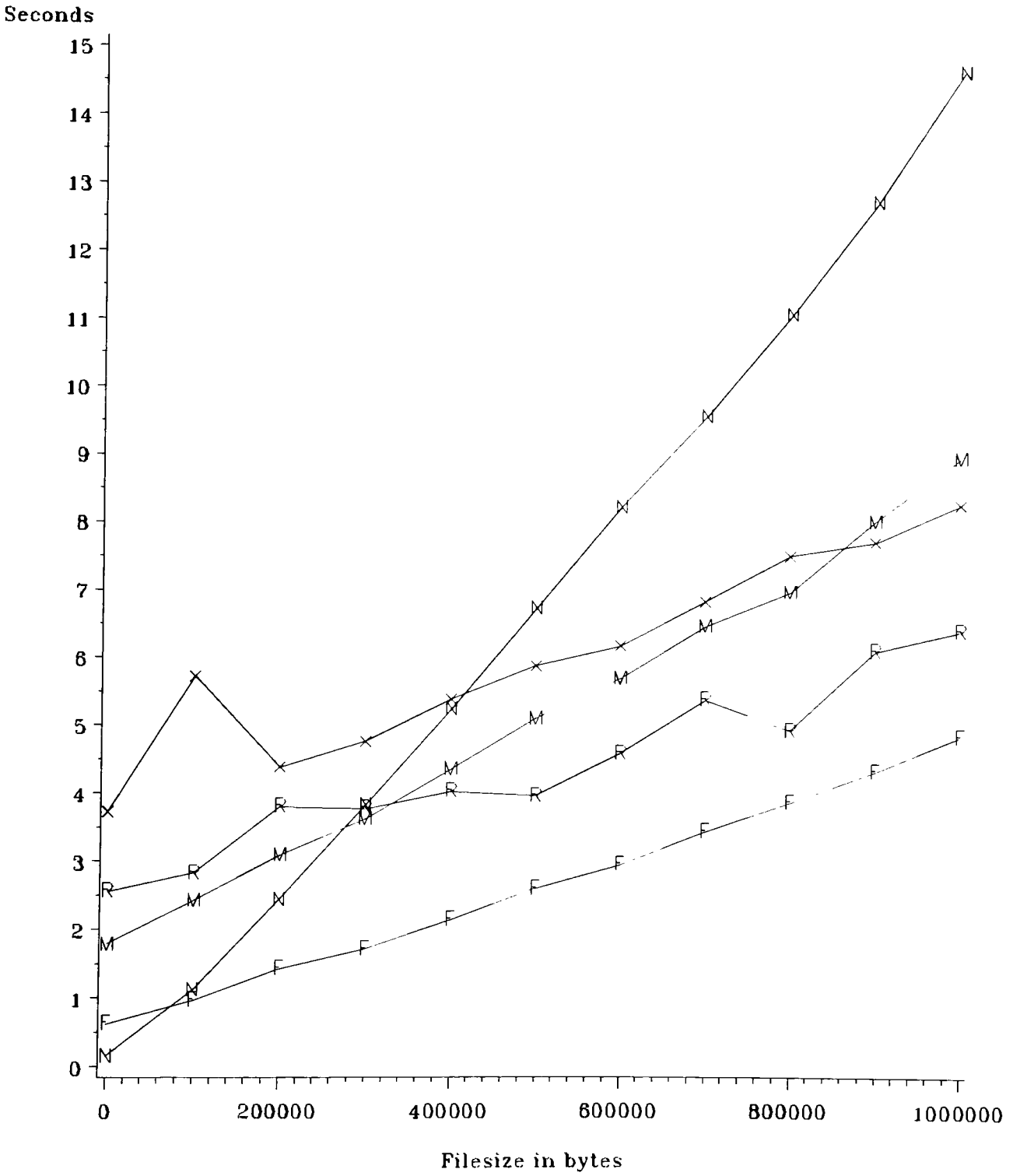


Figure 15

Performance of Filing Protocols
Client Reading from Server

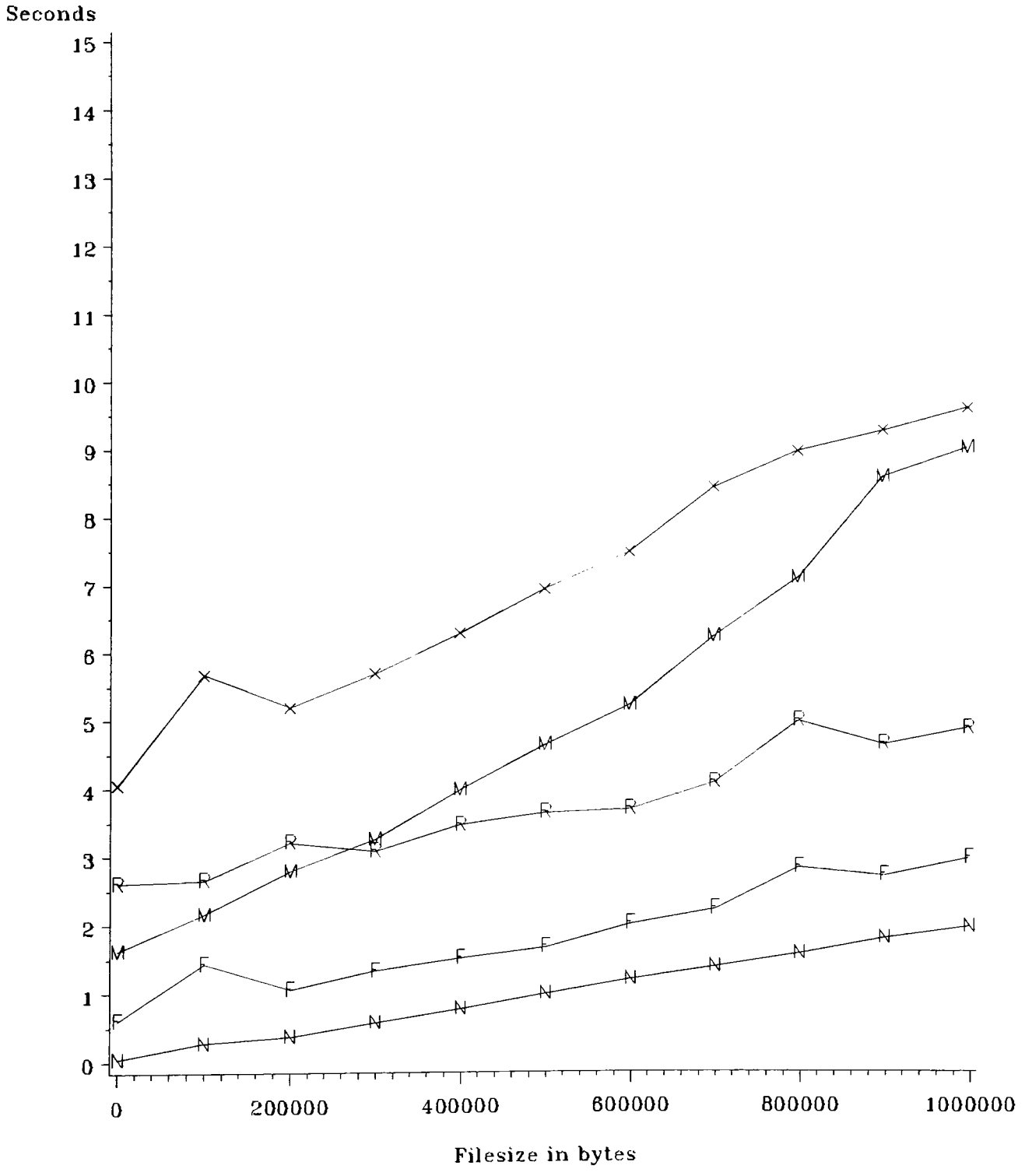


Figure 16

Symmetry of Filing Protocols
Protocol = FTAM

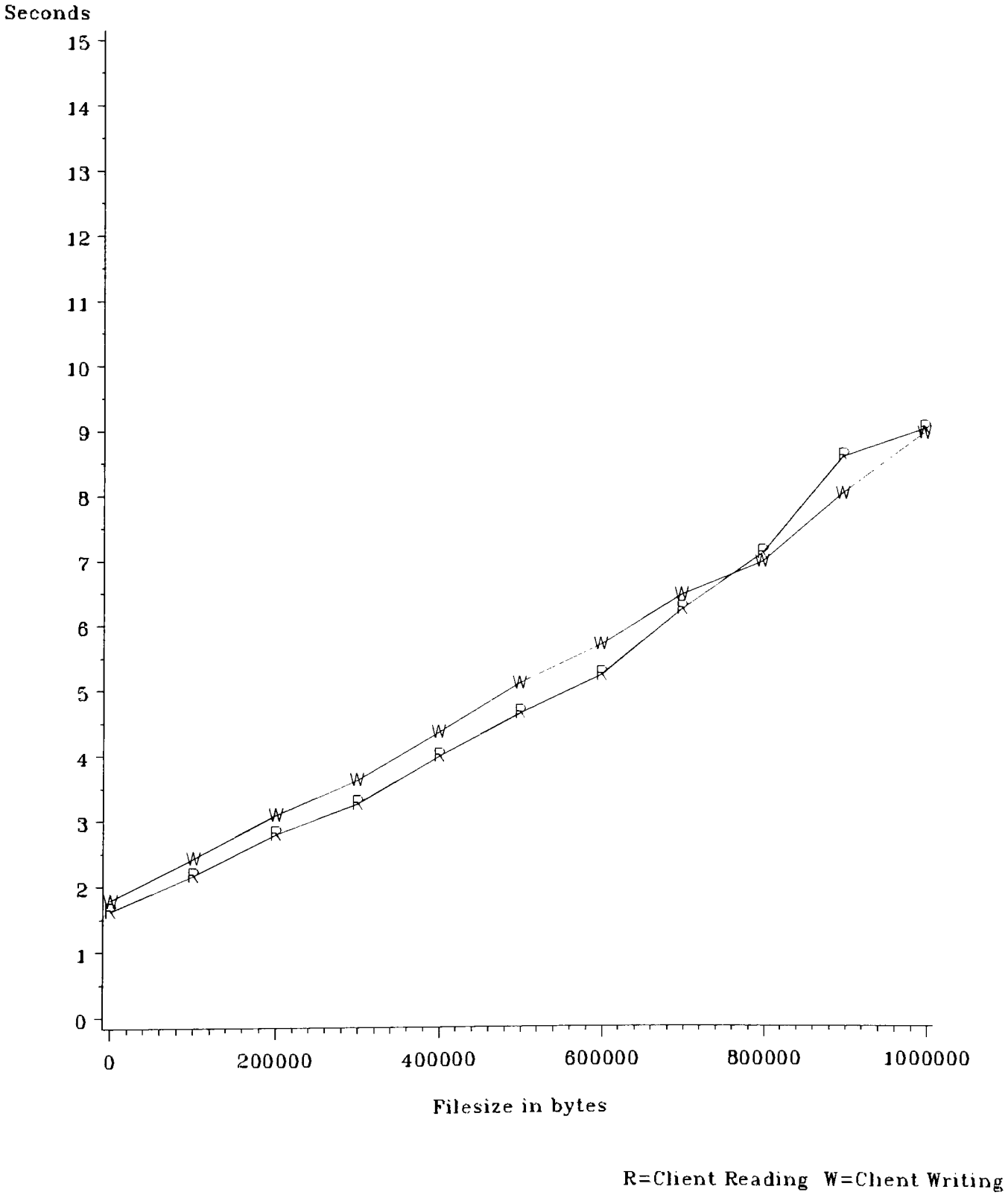


Figure 17

Symmetry of Filing Protocols
Protocol = FTP

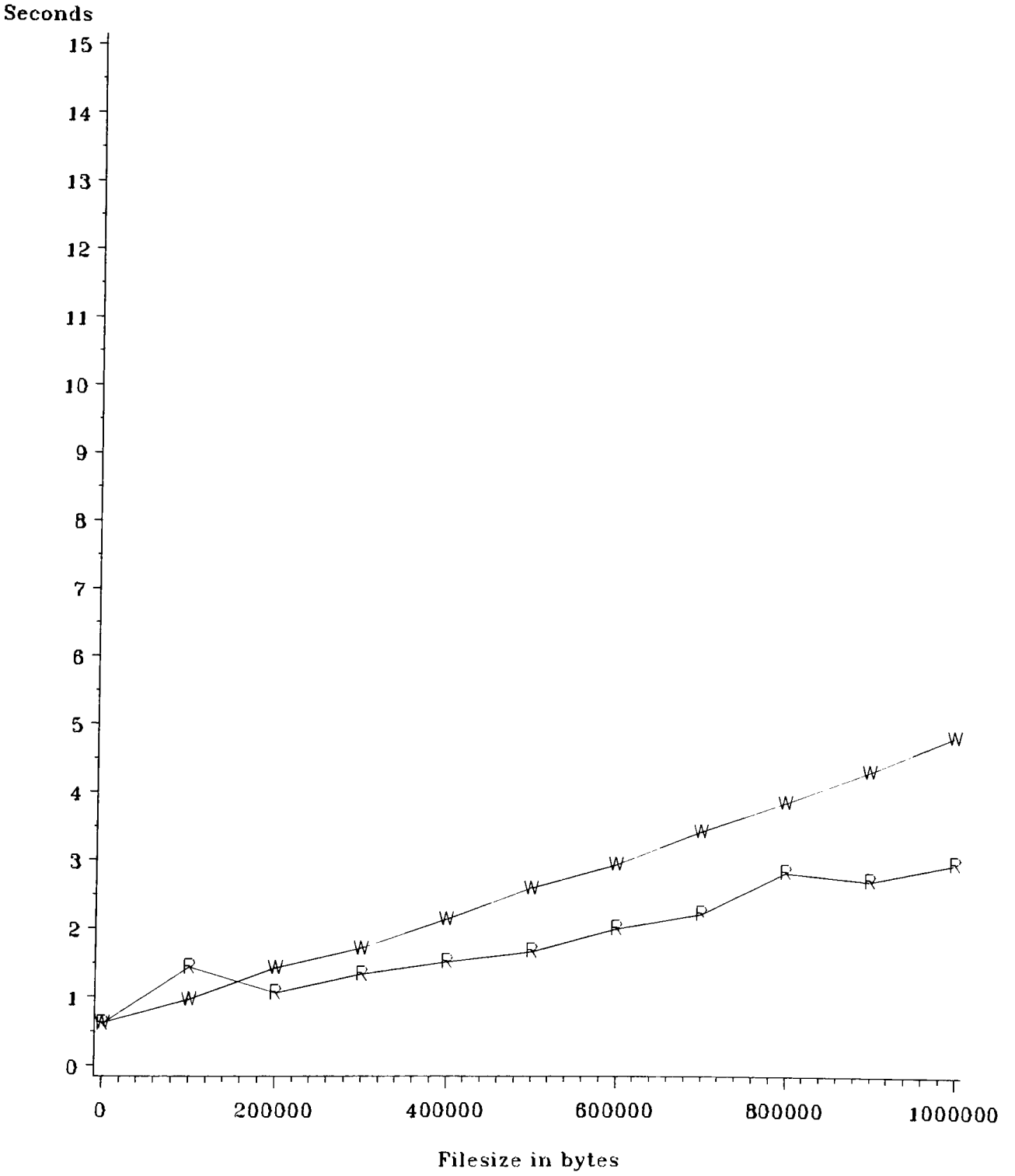


Figure 18

Symmetry of Filing Protocols
Protocol = UNIX rcp

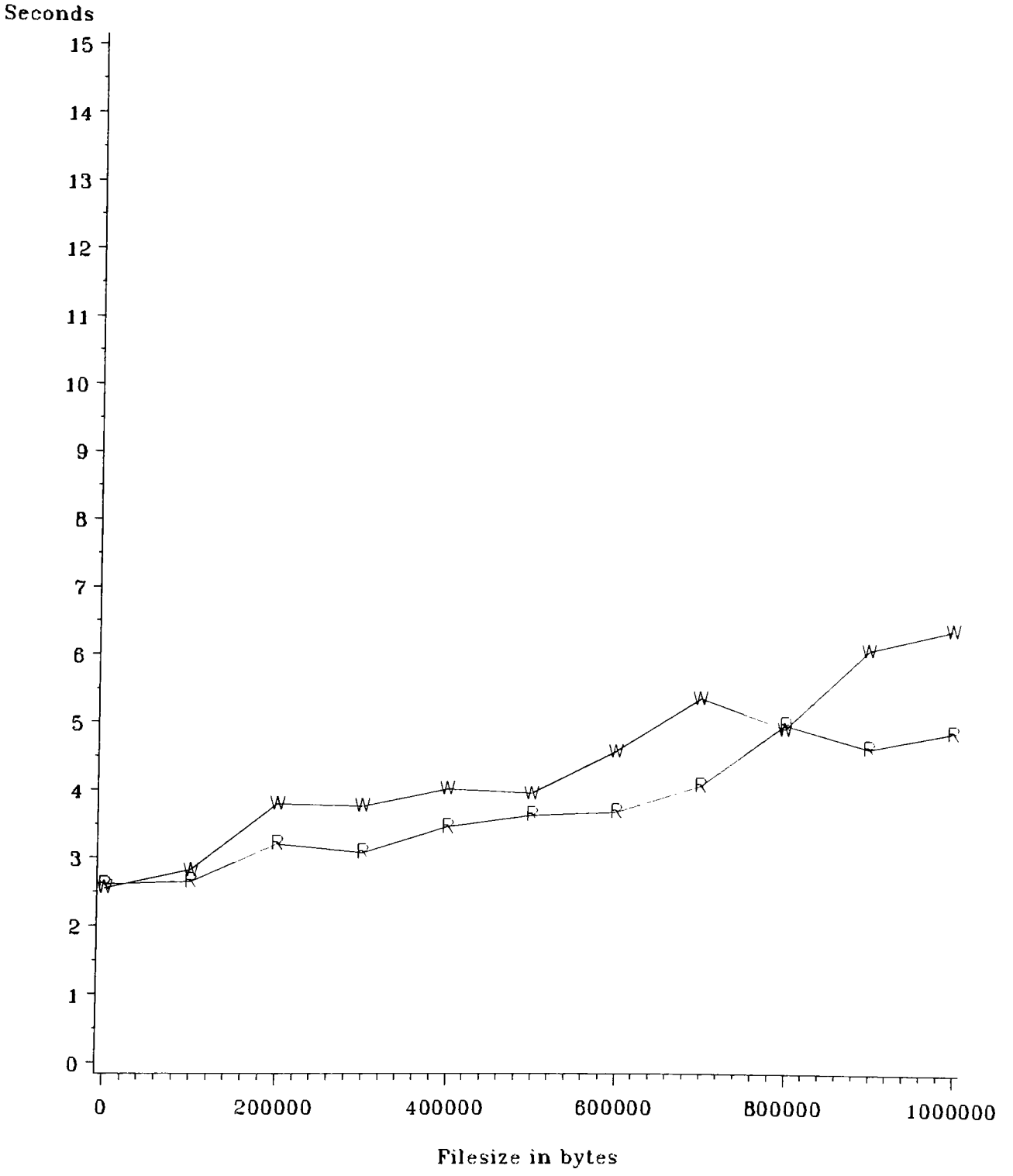


Figure 19

Symmetry of Filing Protocols
Protocol = XNS

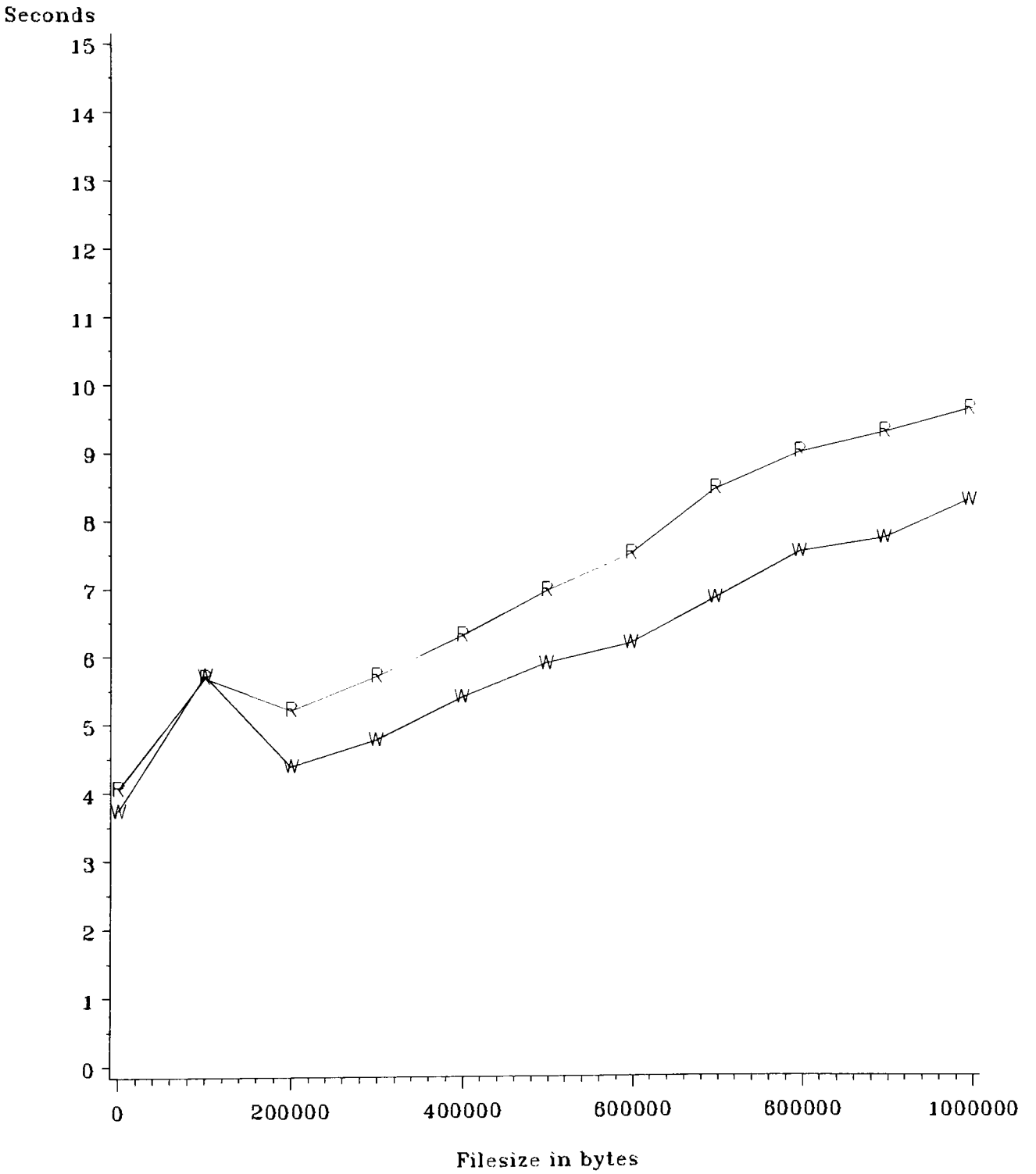


Figure 20

Symmetry of Filing Protocols
Protocol = NFS

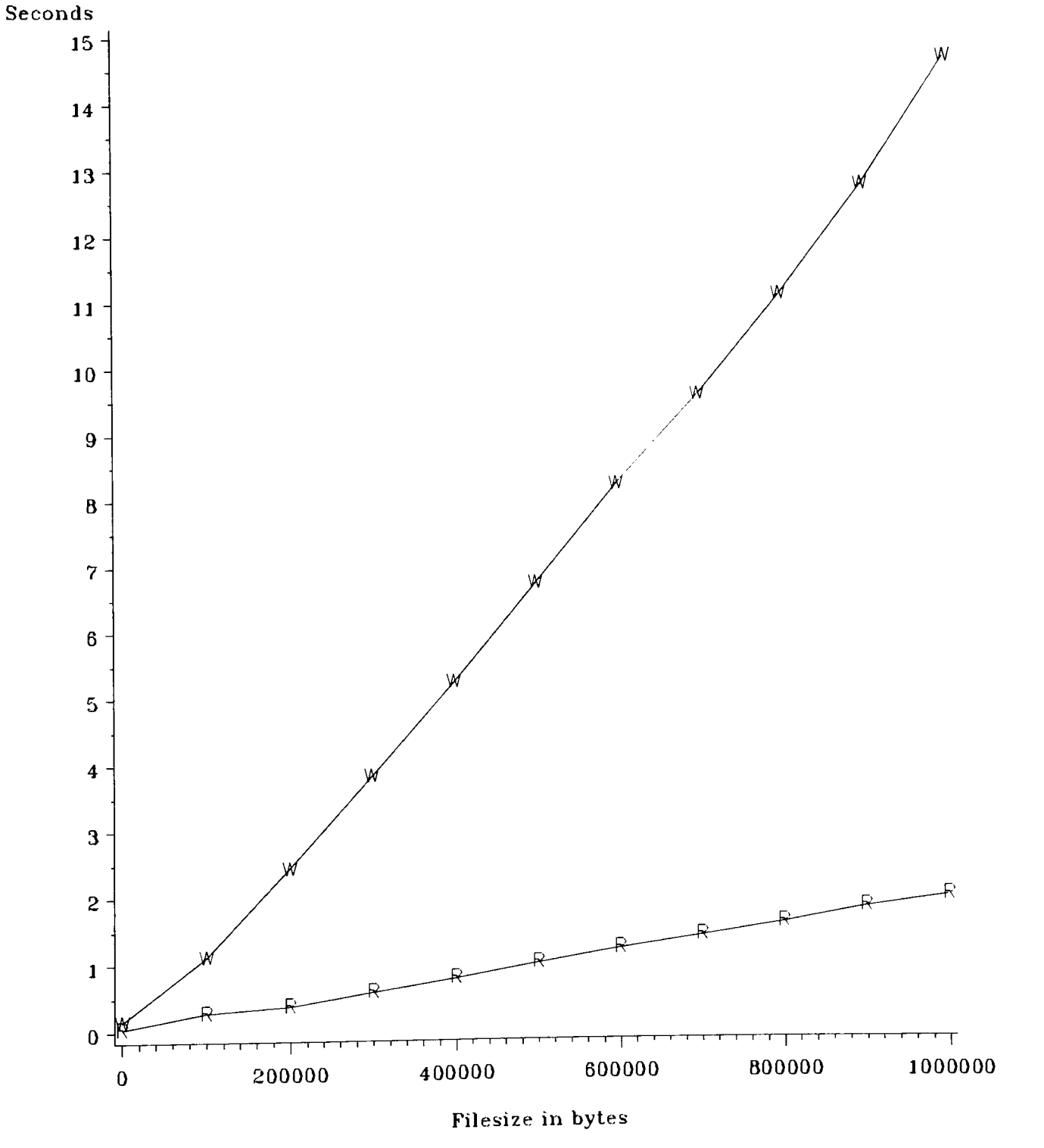


Figure 21

	FTAM	FTP	UNIX rcp	XNS Filing	NFS
protocol symmetry in UNIX implementation	1st place (most symmetrical)				5th place (least symmetrical)
WRITE transfer speed in UNIX implementation	3rd place	1st place (fastest)	2nd place	4th place	5th place (slowest)
READ transfer speed in UNIX implementation	4th place	2nd place	3rd place	5th place (slowest)	1st place (fastest)

Figure 22. Comparison of Performance

4.0 Conclusions

FTAM, FTP, UNIX rcp, XNS Filing, and NFS were each designed with different goals and for different target domains. Consequently, they all have different strengths and weaknesses. FTAM permits users to share file data regardless of the users' hardware or software differences. FTP provides a simple solution for reliably transferring entire files in a TCP/IP environment, shielding the user from variations in file storage systems among hosts. UNIX rcp copies files between trusted UNIX hosts. XNS Filing optimizes file management for a typical office, technical, or university campus environment. NFS provides a seamless extension of file systems so that workstations in a distributed workgroup setting may transparently share data.

FTAM's advantages include its international stature, its strong data translation abilities, and its balanced support for filing operations. Its charter is quite broad: to support whole file transfer between mainframes as well as extending a file system across a network. There are certainly special-purpose protocols that can perform some of these tasks more efficiently than FTAM, as we have seen in this study. Regardless, FTAM has promise as a general-purpose, low-performance mechanism for file service.

FTP is a comparatively simple filing protocol which provides support for diverse operating systems. This simplicity means that it is easy to implement and that it should be expected to have good response times. My measurements confirm this. FTP's greatest weakness is that it not portable to non-TCP/IP environments.

The NFS protocol is designed to be operating system independent, but it was designed in a UNIX environment. Consequently, it has some features which are very "UNIXish". This is its greatest liability. All its implementations treat files exactly as the UNIX operating system treats files. This requirement complicates the porting of NFS into non-UNIX environments. Likewise, rcp is very deeply rooted in the UNIX operating system. Similar problems are not found with FTAM, FTP, and XNS Filing.

NFS is file access oriented, but supports limited file transfer and file management. FTP and rcp are file transfer oriented with no network file extension abilities and limited file management capabilities. FTAM and XNS

Filing sit on the fence between the three, supporting file access, file transfer, and file management. The ISODE implementation of FTAM has not implemented the full richness of the file access and file management part of the protocol specification.

UNIX rcp does not have error recovery capabilities in the event of a machine crash. The clients of FTAM and XNS have a set of error recovery routines to assist in detecting the server crash and rebuilding the server's state when it comes back up. FTP has a similar set of routines for those implementations handling block and compressed transfer modes. Many implementations do not implement these modes and consequently give up all error recovery capabilities. NFS has the most unique approach to error recovery: stateless servers. A NFS client can always assume that its requested operation has completed as soon as the call has returned. Therefore, no error recovery capabilities are required.

There are many similarities between XNS and FTAM. Both view a file as a body of data consisting of attributes and content. Both have a similar exported interface, despite FTAM not using a remote procedure call model. Both use file locking mechanisms to control concurrent access to its files. Both use access control lists to protect its files from unauthorized users. It comes as no surprise that XNS was "part of the inspiration for the OSI model's designers." [DNeib89]

Although NFS and UNIX rcp are based on entirely different models, some minor similarities exist between the two. Both are very heavily rooted in the UNIX operating system, and neither has a stated minimal implementation. Concurrency and Access Control issues are left up to the implementation, and because neither is a session-based protocol, there is no need to specify a password encryption strategy.

FTAM recognizes that a great many types of files exist and it transfers them accordingly. It even has provisions to define new file types. FTP and XNS Filing recognize the most common types of files. This should be adequate for a large percentage of the file types, but it is not as comprehensive as FTAM's file type recognition. NFS and UNIX rcp have no built-in file conversion capabilities. In both protocols, the raw file bit stream is copied without any translation of the file data. Consequently, it is possible for a destination end

system to misunderstand the received data unless the users have prior knowledge of the differences so a conversion utility can be applied.

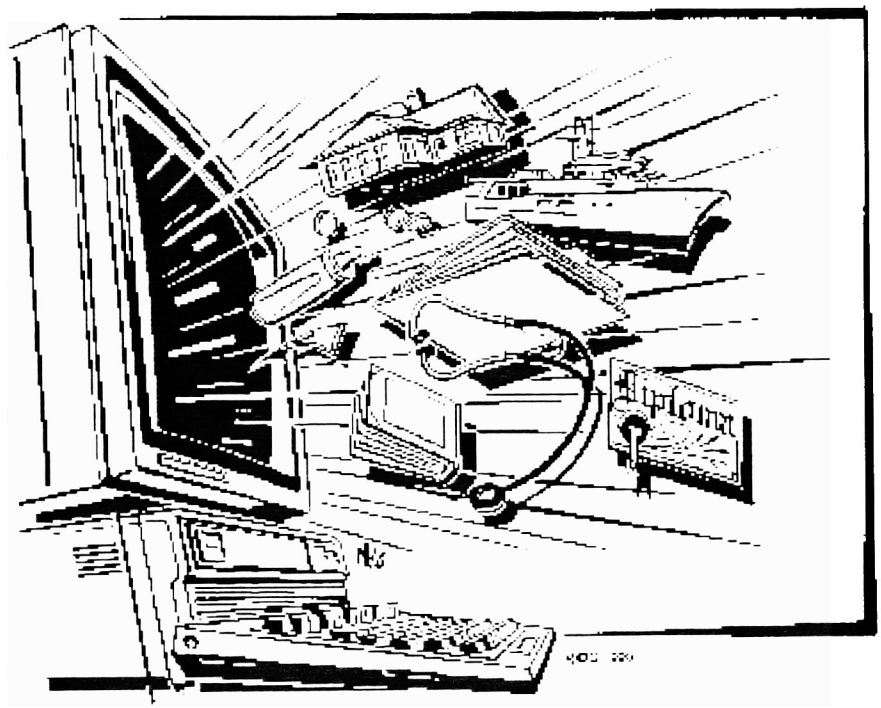
During third party transfers, two servers are controlled by one client. The client governs the transfer of data between the two servers without the data passing through the client. FTP, UNIX rcp, and XNS define and support third party transfers. No third party transfers are allowed in NFS or FTAM.

Both FTAM and XNS Filing are noticeably slower than FTP and NFS read operations. Regardless, good performance should not be the only criteria for selecting a protocol. The FTAM implementation has poor performance, since FTAM attempts to accommodate a variety of file systems. This means more parameters have to be exchanged and negotiated, resulting in slower performance. For network file system operations, FTAM uses a connection-oriented model in which at most a single file may be selected at a given instant.

In most computing environments, two types of filing operations are required: one for general-purpose transfers over different types of networks, and one to extend a file system across a network. NFS is the best choice for use in a UNIX environment when it is necessary to distribute files transparently across a single local area network. FTP is the best choice in a distributed TCP/IP wide area network environment. FTAM is the protocol of choice in an environment with diverse hardware and software platforms.

5.0 Acknowledgements

The author wishes to thank the thesis committee for their guidance, support, and patience. Committee members are Susie Armstrong, Xerox Corporation; Pete Crean, Xerox Corporation; Jim Heliotis, Rochester Institute of Technology; and Cork Russell, Xerox Corporation.



6.0 References

[ASieg89] Alex Siegel, et al. "Deceit: A Flexible Distributed File System." Cornell University. Nov. 1989.

[ATane88] Andrew S. Tanenbaum. Computer Networks 2nd ed. Prentice-Hall, Inc., 1988.

[AWeav90] Alfred Weaver, et al. "Evaluation of Reliable Multicast Protocols." Transfer 1990: 5.

[BKern78] Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language. Prentice-Hall, Inc., 1978.

[BLyon85] Bob Lyon and Gary Sager, et al. "Overview of the Sun Network File System." Sun Microsystems, Inc., January 1985.

[BMetc87] Bob Metcalfe. My Second Network Station. Xerox PARC Forum Video, 23 June 1987

[CHedr89] Charles Hedrick. "Internet Protocols In Depth." Sun Software Technical Bulletin 1989-01.

[CMala90a] Carl Malamud. "Sharing the Wealth: RPCs Help Programs Go Places." Data Communications 21 June 1990: 61.

[CMala90b] Carl Malamud. "Streams Helps Comm-based Services Go with the Flow." Data Communications May 1990: 129.

[CMans89] Carl Manson and Ken Thurber. "Remote Control." Byte July 1989: 235.

[CMoor89] Christopher W. Moore and Marshall T. Rose. "Practical Perspectives on OSI Networking." Oct. 1989.

[DBogg80] David R. Boggs, et al. "Pup: An Internetwork Architecture." Xerox Corporation, July 1979.

[DClark89] David D. Clark, et al. "An Analysis of TCP Processing Overhead." IEEE Communications June 1989: 23.

[DCome88] Douglas Comer. Internetworking with TCP/IP. Prentice-Hall, Inc., 1988.

[DDN85] File Transfer Protocol, DDN Protocol Handbook Volume Two, ARPA RFC 959. DDN Network Information Center, SRI International. December 1985.

[DFinl89] Doug Finlay. "Will XTP resolve bottlenecks for next-generation LANs?" Systems Integration July 1989.

[DMeye90] Douglas Meyer and George Zobrist. "TCP/IP versus OSI." IEEE Potentials February 1990: 16.

[DNeib89] Dale Neibaur. "Understanding XNS: The prototypical internetwork protocol." Data Communications 21 September 1989: 43.

[DSimp89a] David Simpson. "The STREAMS Machine." Mini-Micro Systems. February 1989: 62.

[DSimp89b] David Simpson. "OSI Still a Hard Sell." Systems Integration August 1989: 47.

[EFlin86] Ed Flint. "A study of the Xerox XNS Filing Protocol as implemented on several heterogeneous systems." Thesis: Rochester Institute of Technology, 1986.

[FGrec90] Frank D. Greco. "A Net Gain: NFS, Network File System." Programmer's Journal January/February 1990: 32.

[GBoch80] Gregor V. Bochmann and Carl A. Sunshine. "Formal Methods in Communication Protocol Design." IEEE Transactions on Communications April 1980.

[GChorn78] G. E. Chorn, et al. "The Standard File Transport Protocol." Los Alamos Scientific Laboratory, August 1978.

[ISO88a] *"Information processing systems -- Open Systems Interconnection -- File Transfer, Access and Management, Part 1: General Introduction, ISO 8571-1."* International Organization for Standardization, 1988.

[ISO88b] *"Information processing systems -- Open Systems Interconnection -- File Transfer, Access and Management, Part 2: Virtual Filestore Definition, ISO 8571-2."* International Organization for Standardization, 1988.

[ISO88c] *"Information processing systems -- Open Systems Interconnection -- File Transfer, Access and Management, Part 3: File Service Definition, ISO 8571-3."* International Organization for Standardization, 1988.

[ISO88d] *"Information processing systems -- Open Systems Interconnection -- Protocol Specification for the Association Control Service Element, ISO 8650."* International Organization for Standardization, 1988.

[JCorb89] John Corbin and Chris Silveri. "Open Network Programming." UNIX World December 1989: 115.

[JGill83] James J. Gill. "HASP Encapsulation of File Transfer Protocol (FTP)." Xerox Corporation, March 1983.

[JMogu86] Jeffrey Mogul. "The Leaf File Access Protocol." Stanford University, Dec. 1986.

[JOnio89] Julian Onions. "ISODE: In Support of Migration." Computer Networks and ISDN Systems 10 October 1989: 362.

[JRomk89a] John Romkey. "Networking with BSD-Style Sockets." UNIX World July 1989: 95.

[JRomk89b] John Romkey. "Programming with BSD-Style Sockets," UNIX World August 1989: 97.

[JShoc82] John Shoch and Ed Taft. "Pup File Transfer Protocol Specifiaction-5th edition." Xerox Corporation, 25 May 1982.

[KMarz88] Keith Marzullo and Frank Schmuck. "Supplying High Availability with a Standard Network File System." Cornell University, 1988.

[LMant89] Lee Mantelman. "Opening the file on FTAM." Data Communications November 1989.

[MBrow84] Mark Brown, et al. "The Alpine File System." Xerox Palo Alto Research Center, October 1984.

[MGien78] Michel Gien. "A File Transfer Protocol (FTP)." Computer Networks September/October 1978: 312.

[MPado89] Michael Padovano. "How NFS and RFS compare." Systems Integration December 1989: 159: 27.

[MRose86] Marshall T. Rose and Dwight E. Cass. "OSI Transport Services on top of the TCP." Computer Networks and ISDN Systems Vol 12 No 3 1986: 159.

[MRose90a] Marshall T. Rose. "The ISO Development Environment: User's Manual Volumes 1-4." Performance Systems International, Inc. 12 January 1990.

[MRose90b] Marshall Rose. The Open Book. Prentice-Hall, Inc., 1990.

[MSaty89] M. Satyanarayanan. "A Survey of Distributed File Systems, CMU-CS-89-116." Carnegie Mellon University, February 1989.

[MWood88a] Mark Wood. "Network Performance." Xerox Corporation, 12 August 1988.

[MWood88b] Mark Wood. "Observations on Writing a File Transfer Utility." Xerox Corporation, 12 August 1988.

[NBoly90] Nelson Bolyard. "Protocol Description: BSD 4.3 rcp." Silicon Graphics Inc. October 1990.

[NPete85] Niels K. Peterson and Tom Skovgaard. "Anticipating the ISO File Transfer Standards in an Open Systems Implementation." Computer Networks and ISDN Systems Volume 9 1985: 267.

[NSten76] N. V. Stenning, "A data transfer protocol." Computer Networks Spetember 1976: 99.

[PGree80] Paul E. Green, Jr. "An Introduction to Network Architectures and Protocols." IEEE Transactions on Communications April 1980: 413.

[PGunn89] Per Gunningberg, et al. "Application Protocols and Performance Benchmarks." IEEE Communications June 1989: 30.

[PLini84] Peter F. Linington. "The Virtual Filestore Concept." Computer Networks February 1984: 13.

[PLini89] Peter F. Linington. "File Transfer Protocols." IEEE Journal on Selected Areas in Communication September 1989: 1052.

[RFarr89] Rik Farrow. "Riding the Ethers, from UUCP to NFS." UNIX World November 1989: 81.

[RSand89] Russel Sandberg. *"The Sun Network File System: Design, Implementation and Experience."* Sun Microsystems, Inc., 1989.

[RStev90] W. Richard Stevens. UNIX Network Programming. Prentice-Hall, Inc., 1990.

[RTobi90] Randall L. Tobias. "Telecommunications in the 1990s." Business Horizons January/February 1990: 81.

[SAS85a] *SAS User's Guide: Basics Version 5 Edition.* SAS Institute Inc., 1985.

[SAS85b] *SAS/GRAPH User's Guide Version 5 Edition.* SAS Institute Inc., 1985.

[SChan88] Samuel T. Chanson and Mei-Jean Goh. "Implementation of the ISO File Transfer, Access and Management Protocol in the UNIX Environment." IEEE 1988: 44.

[SFish90] Sharon Fisher. "In the Trenches with Communications Protocols." UNIX World Supplement: Connectivity 1990: 2.

[SPrat85] Stephen Prata. Advanced UNIX -- A Programmer's Guide. Howard W. Sams and Company, 1985.

[Sun86a] *Inter-Process Communication Primer.* Sun Microsystems, Inc., 1986.

[Sun86b] *Networking on the Sun Workstation: Network File System Protocol Specification Revision B.* Sun Microsystems, Inc., 17 February 1986.

[Sun86c] *Networking on the Sun Workstation: Network Services Guide.* Sun Microsystems, Inc., February 1986.

[Sun90] Distributed Computing Road Map: The Future of Open Network Computing. Sun Microsystems, Inc., May 1990.

[VSten86] Vic Stenning. "A data transfer protocol." Computer Networks September 1986.

[WBlac89] William Black. "File Access and Serving in an OSI Environment." Computer Networks and ISDN Systems 10 October 1989: 294.

[WDall90] William Dallas. "A digital prescription for X-ray overload." IEEE Spectrum April 1990.

[WStall88] William Stallings. "The Glue for Internetworking." Byte LAN Supplement 1988: 221.

[Xerox75] *Pup File Transfer Protocol Specification, 5th Edition.* Xerox Corporation, 25 May 1982.

[Xerox81a] *Filing Protocol, Xerox Network Systems Standard 108605.* Xerox Corporation, December 1981.

[Xerox81b] *Courier: The Remote Procedure Call Protocol, Xerox Network Systems Standard 038112.* Xerox Corporation, December 1981.

[Xerox85] *Xerox Network Systems Architecture General Information Manual.* Xerox Corporation, 1985.

[Xerox86] *Authentication Protocol, Xerox Network Systems Standard 098605.* Xerox Corporation, May 1986.

[Xerox89a] *XNS for UNIX V.3 Porting Guide, XSI 880314.* Xerox Corporation, March 1989.

[Xerox89b] *XNS for UNIX V.3 Programming Guide XSI 880414.* Xerox Corporation, April 1989.

[Xerox89c] *XNS for UNIX V.3 User Guide XSI 880213.* Xerox Corporation, April 1989.

Appendix A

	FTAM	FTP	UNIX rcp	XNS Filing	NFS
formal protocol specification exists	Yes	Yes	No	Yes	Yes
filing protocol class	general-purpose file transfer system	general-purpose file transfer system	general-purpose file transfer system	general-purpose file transfer system	network file system
RPC-based	No	No	No	Yes, uses Courier	Yes, uses Sun RPC
machine-independent data representation	Yes, uses ASN.1 and BER	Yes, built into FTP	No	Yes, uses object layer of Courier	Yes, uses XDR
session-based protocol	Yes	Yes	No	Yes	No
preferred transport protocol	TP	TCP	TCP	SPP	UDP
type of transport	connection oriented	connection oriented	connection oriented	connection oriented	connectionless
number of network connections required by the protocol	1	2	1	1	0
separate mechanism for bulk data	Yes	Yes	No	Yes	No
files represented by attributes and content	Yes	No	No	Yes	Yes
third party transfers	No	Yes	Yes	Yes	Yes
approximate date of first implementation	late 1980s	late 1970s	mid 1980s	early 1980s	mid 1980s
exported interface	30 primitives and > 30 parameters	33 commands and 12 arguments	the rcp command itself	23 remote procedures and 14 parameter data types	17 procedures and 11 parameter data types

Summary of Protocol Comparisons

	FTAM	FTP	UNIX rcp	XNS Filing	NFS
stated minimal implementation	Yes	Yes	No	Yes	Yes
concurrency control mechanisms	uses 4 file locks: not required, shared, exclusive, and no access	implementation dependent	implementation dependent	uses 2 file locks: share and exclusive	implementation dependent
file-level locks	Yes	not applicable	not applicable	Yes	not applicable
record-level locks	Yes	not applicable	not applicable	No	not applicable
access control	uses ACL mechanism	implementation dependent	implementation dependent	uses ACL mechanism	implementation dependent
password handling	implementation dependent	implementation dependent	not applicable	always encrypted	not applicable
error recovery	uses checkpoint and docket mechanism	none for stream mode; uses checkpoint mechanism for block and compressed mode	none	remote procedure attempts to undo its operations	none required because of stateless server model
protocol symmetry in UNIX implementation	first place (most symmetrical)				fifth place (least symmetrical)
WRITE transfer speed in UNIX implementation	third place	first place (fastest)	second place	fourth place	fifth place (slowest)

Summary of Protocol Comparisons

	FTAM	FTP	UNIX rcp	XNS Filing	NFS
READ transfer speed in UNIX implementation	fourth place	second place	third place	fifth place (slowest)	first place (fastest)

Summary of Protocol Comparisons

Appendix B

Source code listing for the timing routines that were added to each protocol's implementation.

The code below declares the timing variables and starts the timer. This code was added at the beginning of the main routine:

```
/* EM begin new */
struct timeval start, stop, elapsedtime;
float s;
/* end new */

/* EM begin new */
gettimeofday(&start, (struct timezone*)0);
/* EM end new */
```

The code below turns off the timer and performs the subtraction to determine the elapsed time. This code was added at the end of the main routine:

```
/* EM new */
gettimeofday(&stop, (struct timezone *)0);
tvsub(&time, stop, start);
s=elapsedtime.tv_sec + (elapsedtime.tv_usec / 1000000.);
printf("%.3g\n", s);
/* EM end new */

/* EM begin new*/
tvsub(tdiff, t1, t0)
        struct timeval *tdiff, *t1, *t0;
{
        tdiff->tv_sec   t1->tv_sec   t0->tv_sec;
        tdiff->tv_usec   t1->tv_usec   t0->tv_usec;
        if (tdiff->tv_usec < 0)
                tdiff->tv_sec--, tdiff->tv_usec += 1000000;
}
/* EM end new */
```

UNIX shell scripts measuring file transfer performance:

A UNIX shell script was built to perform all 1100 measurements (5 protocols X 2 directions X 10 transfers X 11 filesizes). The top level script, named *allofem*, calls a script for each of the five protocols, which in turn calls lower level scripts, as illustrated in Figure 16. Source code listings of the shell scripts appear in this Appendix, as well as an example output file. The output file from *allofem* is used as the input to SAS. The first SAS procedure reads the data, drops the fastest and slowest transfer, calculates averages, and organizes the data into permanent SAS data sets. A second SAS procedure uses the permanent data sets to generate the six plots included in Section 3.5 of this paper.

The shell scripts used for measuring FTAM, FTP, and XNS Filing are very similar, as these protocols all use a session-based model. The only major difference between each of these protocols' measurement routines is the bottom-level script, *xxx*script, where *xxx* is the name of the protocol. To avoid redundancy, the full script set for FTAM is listed in this appendix, and only the bottom-level scripts for FTP and XNS Filing are listed. Likewise, the script sets for NFS and UNIX rcp are almost identical, so only NFS's scripts are listed in this appendix.

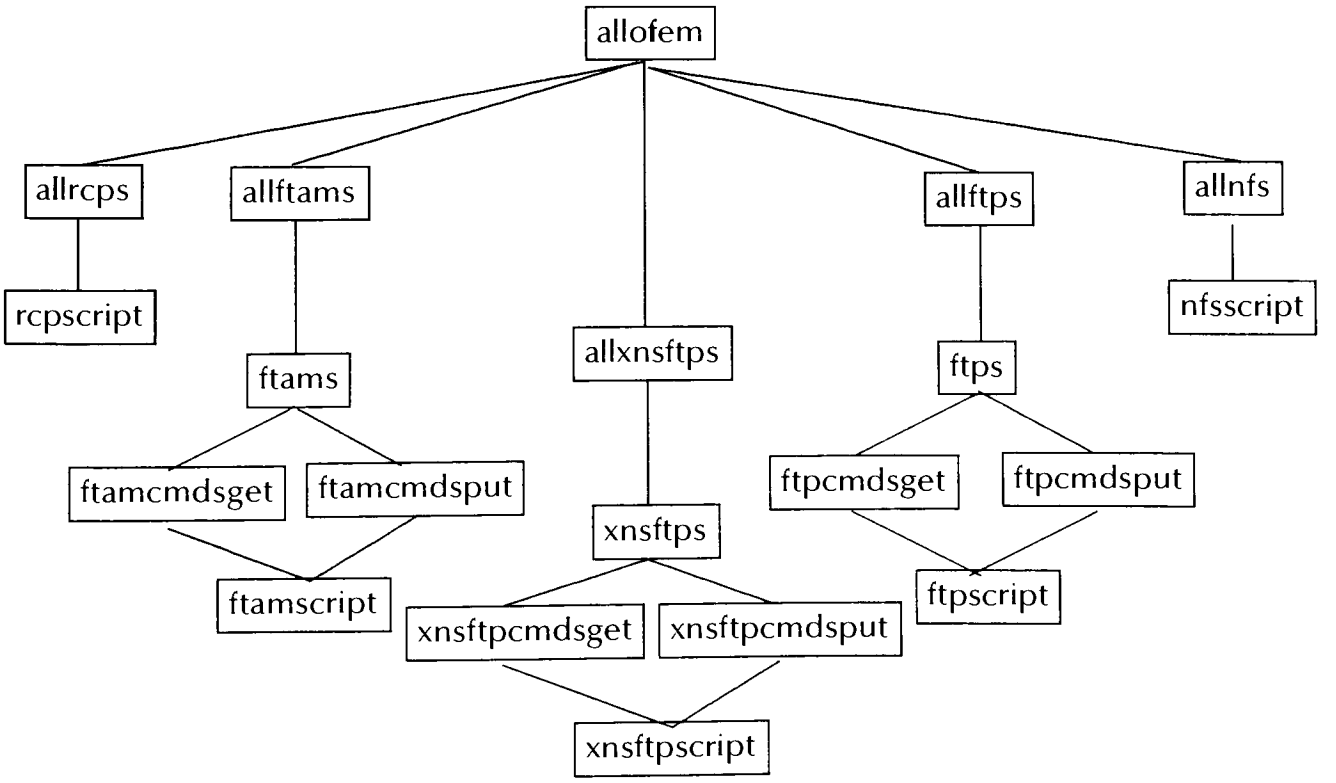


Figure 23. Relationship of the UNIX shell scripts that measure implementation performance

File: *allofem*

```
#! /bin/csh
# usage:  allofem
# description:       top level script of the script set that measures
#                    file transfer performance for ftam, ftp, rcp, xns,
#                    and nfs protocols.
# 4/91 elayne mcfaul
#
cd ftam
allftams
cd ../ftp
allftps
cd ../rcp
allrcps
cd ../xnsftp
allxnsftps
cd ../nfs
allnfs
```

FTAM Measurements

File: *allftams*

```
#! /bin/csh
# usage:  allftams
# description:       part of the script set that measures file
#                    transfer performance.
#
#                    top level ftam script that measures performance
#                    for 11 file sizes.
# 4/91 elayne mcfaul
#
foreach filesize (1 100000 200000 300000 400000 500000 600000 700000 800000 900000 1000000)
                  sh ftams $filesize get
```

```
                sh ftams $filesize put
end
```

File: *ftams*

```
#! bin/sh
# usage:  sh ftams filename direction (put or get)
# description:       part of the script set that measures ftam file
#                    transfer performance.
#
#                    cleans up source and destination directories and
#                    sets up the files to be transferred.
# 4/91 elayne mcfaul
#
# housekeeping
rm -f /home/jade/mcfaul/testers/a
rm -f /home/jade/mcfaul/testers/b
rm -f /home/jade/mcfaul/testers/c
rm -f /home/jade/mcfaul/testers/d
rm -f /home/jade/mcfaul/testers/e
rm -f /home/jade/mcfaul/testers/f
rm -f /home/jade/mcfaul/testers/g
rm -f /home/jade/mcfaul/testers/h
rm -f /home/jade/mcfaul/testers/i
rm -f /home/jade/mcfaul/testers/j
rm -f /home/smokey/mcfaul/testers/a
rm -f /home/smokey/mcfaul/testers/b
rm -f /home/smokey/mcfaul/testers/c
rm -f /home/smokey/mcfaul/testers/d
rm -f /home/smokey/mcfaul/testers/e
rm -f /home/smokey/mcfaul/testers/f
rm -f /home/smokey/mcfaul/testers/g
rm -f /home/smokey/mcfaul/testers/h
rm -f /home/smokey/mcfaul/testers/i
rm -f /home/smokey/mcfaul/testers/j
#
```

```
echo FTAM
echo $1
# set up files
case $2 in
                get)  echo GET
                      cp /usr/rtcc/mcfaul/testers/$1 /home/jade/mcfaul/testers/a
                      cp /usr/rtcc/mcfaul/testers/$1 /home/jade/mcfaul/testers/b
                      cp /usr/rtcc/mcfaul/testers/$1 /home/jade/mcfaul/testers/c
                      cp /usr/rtcc/mcfaul/testers/$1 /home/jade/mcfaul/testers/d
                      cp /usr/rtcc/mcfaul/testers/$1 /home/jade/mcfaul/testers/e
                      cp /usr/rtcc/mcfaul/testers/$1 /home/jade/mcfaul/testers/f
                      cp /usr/rtcc/mcfaul/testers/$1 /home/jade/mcfaul/testers/g
                      cp /usr/rtcc/mcfaul/testers/$1 /home/jade/mcfaul/testers/h
                      cp /usr/rtcc/mcfaul/testers/$1 /home/jade/mcfaul/testers/i
                      cp /usr/rtcc/mcfaul/testers/$1 /home/jade/mcfaul/testers/j
                      ftamcmdsget;;
                put)  echo PUT
                      cp /usr/rtcc/mcfaul/testers/$1 /home/smokey/mcfaul/testers/a
                      cp /usr/rtcc/mcfaul/testers/$1 /home/smokey/mcfaul/testers/b
                      cp /usr/rtcc/mcfaul/testers/$1 /home/smokey/mcfaul/testers/c
                      cp /usr/rtcc/mcfaul/testers/$1 /home/smokey/mcfaul/testers/d
                      cp /usr/rtcc/mcfaul/testers/$1 /home/smokey/mcfaul/testers/e
                      cp /usr/rtcc/mcfaul/testers/$1 /home/smokey/mcfaul/testers/f
                      cp /usr/rtcc/mcfaul/testers/$1 /home/smokey/mcfaul/testers/g
                      cp /usr/rtcc/mcfaul/testers/$1 /home/smokey/mcfaul/testers/h
                      cp /usr/rtcc/mcfaul/testers/$1 /home/smokey/mcfaul/testers/i
                      cp /usr/rtcc/mcfaul/testers/$1 /home/smokey/mcfaul/testers/j
                      ftamcmdsput:;
                *)    echo Incorrect syntax ;;
esac
```

File: *ftamcmdsget*

```
#! /bin/csh
# usage:  ftamcmdsget
# description:     part of the script set that measures ftam file
```

```
#                   transfer performance.
#
#                   performs the (get) transfer 10 times with 10 uniquely
#                   named files.
# 4/91 elayne mcfaul
#
foreach file (a b c d e f g h i j)
                    ftamscript $file ../$file get
end
```

File: *ftamcmdsput*

```
#! /bin/csh
# usage:  ftamcmdsput
# description:      part of the script set that measures ftam file
#                   transfer performance.
#
#                   performs the (put) transfer 10 times with 10 uniquely
#                   named files.
# 4/91 elayne mcfaul
#
foreach file (a b c d e f g h i j)
                    ftamscript ../$file $file put
end
```

File: *ftamscript*

```
#! /bin/csh
# usage:  ftamscript filename1 filename2 direction (get or put)
# description:      part of the script set that measures ftam file
#                   transfer performance.
#
#                   bottom level script that invokes ftam and
#                   performs the transfer requested via the script arguments
# 4/91 elayne mcfaul
```

```
#
/home/smokey/comm/isode/isode-6.0/ftam2/xftam << EOF
open jade
mcfaul
set type binary
set realstore unix
cd /home/jade/mcfaul/testers
$3 $1 $2
quit
EOF
```

FTP Measurements

Files: *allftps*, *ftps*, *ftpcmdsget*, *ftpcmdsput* (see FTAM's shell scripts)

File: *ftpscript*

```
#! /bin/csh
# usage:  ftpscript filename1 filename2 direction (get or put)
# description:     part of the script set that measures ftp file
#                  transfer performance.
#
#                  bottom level script that invokes ftp and
#                  performs the transfer requested via the script arguments
# 4/91 elayne mcfaul
#
~mcfaul/ftp/ftpstat/emftp -n jade << EOF
user mcfaul thesis
binary
cd /home/jade/mcfaul/testers
$3 $1 $2
quit
EOF
```

XNS Filing Measurements

Files: *allxnsftps*, *xnsftps*, *xnsftpcmdsget*, *xnsftpcmdsput* (see FTAM's shell scripts)

File: *xnsftpscript*

```
#! /bin/csh
# usage:  xnsftpscript filename1 filename2 direction (get or put)
# description:     part of the script set that measures xns file
#                  transfer performance.
#
#                  bottom level script that invokes xnsftp and
#                  performs the transfer requested via the script arguments
# 4/91 elayne mcfaul
#
/home/smokey/comm/exns/examples/filing←client/xnsftp  -n jade << EOF
user mcfaul turkey
type binary
cd /home/jade/mcfaul/testers
$3 $1 $2
quit
EOF
```

NFS Measurements

File: *allnfs*

```
#! /bin/csh
# usage:  allnfs
# description:     part of the script set that measures file
#                  transfer performance.
#
#                  top level nfs script that measures performance
```

```
#                   for 11 file sizes.
# 4/91 elayne mcfaul
#
foreach filesize (1 100000 200000 300000 400000 500000 600000 700000 800000 900000 1000000)
                    nfsscript $filesize
end
```

File: *nfsscript*

```
#! /bin/csh
# usage:  nfsscript filename
# description:      part of the script set that measures nfs file
#                   transfer performance.
#
#                   bottom level script that cleans up source and
#                   destination directories,and sets up the files to be
#                   transferred.
#                   with 10 uniquely named files, pushes 10 copies over
#                   and pulls 10 copies over.
# 4/91 elayne mcfaul
#
# housekeeping
rm -f /home/jade/mcfaul/testers/a
rm -f /home/jade/mcfaul/testers/b
rm -f /home/jade/mcfaul/testers/c
rm -f /home/jade/mcfaul/testers/d
rm -f /home/jade/mcfaul/testers/e
rm -f /home/jade/mcfaul/testers/f
rm -f /home/jade/mcfaul/testers/g
rm -f /home/jade/mcfaul/testers/h
rm -f /home/jade/mcfaul/testers/i
rm -f /home/jade/mcfaul/testers/j
rm -f /home/smokey/mcfaul/testers/a
rm -f /home/smokey/mcfaul/testers/b
rm -f /home/smokey/mcfaul/testers/c
rm -f /home/smokey/mcfaul/testers/d
```

```
rm -f /home/smokey/mcfaul/testers/e
rm -f /home/smokey/mcfaul/testers/f
rm -f /home/smokey/mcfaul/testers/g
rm -f /home/smokey/mcfaul/testers/h
rm -f /home/smokey/mcfaul/testers/i
rm -f /home/smokey/mcfaul/testers/j
#
# set up files
# pull (get) first
#
cp ~mcfaul/testers/$1 /home/jade/mcfaul/testers/a
cp ~mcfaul/testers/$1 /home/jade/mcfaul/testers/b
cp ~mcfaul/testers/$1 /home/jade/mcfaul/testers/c
cp ~mcfaul/testers/$1 /home/jade/mcfaul/testers/d
cp ~mcfaul/testers/$1 /home/jade/mcfaul/testers/e
cp ~mcfaul/testers/$1 /home/jade/mcfaul/testers/f
cp ~mcfaul/testers/$1 /home/jade/mcfaul/testers/g
cp ~mcfaul/testers/$1 /home/jade/mcfaul/testers/h
cp ~mcfaul/testers/$1 /home/jade/mcfaul/testers/i
cp ~mcfaul/testers/$1 /home/jade/mcfaul/testers/j
#
#
echo NFS
echo $1
echo GET
#
foreach file (a b c d e f g h i j)
                    /home/smokey/mcfaul/emcp /home/jade/mcfaul/testers/$file ../$file
end
#
#
# housekeeping
rm -f /home/jade/mcfaul/testers/a
rm -f /home/jade/mcfaul/testers/b
rm -f /home/jade/mcfaul/testers/c
rm -f /home/jade/mcfaul/testers/d
rm -f /home/jade/mcfaul/testers/e
```

```
rm -f /home/jade/mcfaul/testers/f
rm -f /home/jade/mcfaul/testers/g
rm -f /home/jade/mcfaul/testers/h
rm -f /home/jade/mcfaul/testers/i
rm -f /home/jade/mcfaul/testers/j
rm -f /home/smokey/mcfaul/testers/a
rm -f /home/smokey/mcfaul/testers/b
rm -f /home/smokey/mcfaul/testers/c
rm -f /home/smokey/mcfaul/testers/d
rm -f /home/smokey/mcfaul/testers/e
rm -f /home/smokey/mcfaul/testers/f
rm -f /home/smokey/mcfaul/testers/g
rm -f /home/smokey/mcfaul/testers/h
rm -f /home/smokey/mcfaul/testers/i
rm -f /home/smokey/mcfaul/testers/j
#
# set up files
# push (put) files
#
cp ~mcfaul/testers/$1 /home/smokey/mcfaul/testers/a
cp ~mcfaul/testers/$1 /home/smokey/mcfaul/testers/b
cp ~mcfaul/testers/$1 /home/smokey/mcfaul/testers/c
cp ~mcfaul/testers/$1 /home/smokey/mcfaul/testers/d
cp ~mcfaul/testers/$1 /home/smokey/mcfaul/testers/e
cp ~mcfaul/testers/$1 /home/smokey/mcfaul/testers/f
cp ~mcfaul/testers/$1 /home/smokey/mcfaul/testers/g
cp ~mcfaul/testers/$1 /home/smokey/mcfaul/testers/h
cp ~mcfaul/testers/$1 /home/smokey/mcfaul/testers/i
cp ~mcfaul/testers/$1 /home/smokey/mcfaul/testers/j
#
echo NFS
echo $1
echo PUT
#
foreach file (a b c d e f g h i j)
```

```
                /home/smokey/mcfaul/emcp ../$file /home/jade/mcfaul/testers/$file
end
```

UNIX rcp measurements

Files: *allrcps*, *rcpscript* (see NFS's shell scripts)

Sample Output from *allofem* shell script. This file serves as the input to SAS.

```
FTAM
1
GET
3.51
2.19
1.73
2.48
1.76
1.53
1.6
1.61
1.49
1.64
FTAM
1
PUT
4.6
1.69
1.77
1.7
1.68
1.73
1.79
1.86
6.78
1.66
```

```
FTAM
100000
GET
2.74
2.92
:
rest of data
:
RCP
1000000
GET
4.69
4.48
4.71
4.96
4.72
4.77
5.39
4.65
4.96
4.7
RCP
1000000
PUT
6.9
5.76
6.36
6.17
6.59
6.4
6.42
6.07
6.75
5.98
```

SAS Procedure #1, which reads in and reorganizes all the data:

```
/*******************************************************
Name:  getdata.sas

Description:
getdata.sas reads in the protocol transfer statistics,
performs averages, and reorganizes the data into permanent
SAS data sets for later processing by SAS graphics.

The expected data file is contiguous multiple sets of data, all of
which have the following format:

                    protocol name (FTAM, FTP, RCP, XNS, or NFS)
                    file size in bytes
                    direction (PUT or GET)
                    m1
                    m2
                    :
                    :
                    m9
                    m10

m-1-m10 are real numbers representing the transfer time.


Maintenance:
4/91                Elayne McFaul Original Author
*******************************************************/

option   ls=132;

filename data    '[mcfaul11.sas.thesis]thesis2.dat';
libname  saslib  '[mcfaul11.sas.thesis]';

data work1;
infile data missover;
```

```
input              prot $
                   #2 filesize
                   #3 directn $
                   #4 m1
                   #5 m2
                   #6 m3
                   #7 m4
                   #8 m5
                   #9 m6
                   #10 m7
                   #11 m8
                   #12 m9
                   #13 m10;


min    min (m1,m2,m3,m4,m5,m6,m7,m8,m9,m10);
max    max (m1,m2,m3,m4,m5,m6,m7,m8,m9,m10);
seconds    ((sum ( m1, m2, m3, m4, m5, m6, m7, m8, m9, m10))    max    min) / 8;
proc print data    work1;
                      title 'work1';


data ftamset;
                   set work1;
                   drop m1-m10 min max;
                   if prot    'FTAM';
                   rename seconds=ftam;
proc print data    ftamset;
                      title 'ftamset';


data ftpset;
                   set work1;
                   drop m1-m10 min max;
                   if prot    'FTP';
                   rename seconds=ftp;
proc print data    ftpset;
                      title 'ftpset';
```

```
data rcpset;
                    set work1;
                    drop m1-m10 min max;
                    if prot   'RCP';
                    rename seconds=rcp;
proc print data   rcpset;
                      title 'rcpset';

data xnsset;
                    set work1;
                    drop m1-m10 min max;
                    if prot   'XNS';
                    rename seconds=xns;
proc print data   xnsset;
                      title 'xnsset';

data nfsset;
                    set work1;
                    drop m1-m10 min max;
                    if prot   'NFS';
                    rename seconds=nfs;
proc print data   nfsset;
                      title 'nfsset';

data saslib.alldata;
                    merge ftamset ftpset rcpset xnsset nfsset;
                    by filesize directn;
                    drop prot;
proc print data   saslib.alldata;
                      title 'saslib.alldata';

data saslib.allputs;
                    set saslib.alldata:
                    if directn   'PUT';
proc print data   saslib.allputs;
                      title 'saslib.allputs';
```

```
data saslib.allgets;
                  set saslib.alldata;
                  if directn   'GET';
proc print data   saslib.allgets;
                     title 'saslib.allgets';

data puts;
                  set work1;
                  drop m1-m10 min max;
                  if directn   'PUT';
                  rename seconds=put;
proc print data   puts;
                     title 'puts';

data gets;
                  set work1;
                  drop m1-m10 min max;
                  if directn   'GET';
                  rename seconds=get;
proc print data   gets;
                     title 'gets';

proc sort
                  data   puts;
                  by prot filesize;

proc sort
                  data   gets;
                  by prot filesize;

data putget;
                  merge puts gets;
                  by prot filesize;
                  drop directn;
proc print data   putget;
                     title 'putget';
```

```
data saslib.ftam;
                    set putget;
                    if prot   'FTAM';
proc print data   saslib.ftam;
                       title 'saslib.ftam';


data saslib.ftp;
                    set putget;
                    if prot   'FTP';
proc print data   saslib.ftp;
                       title 'saslib.ftp';


data saslib.rcp;
                    set putget;
                    if prot   'RCP';
proc print data   saslib.rcp;
                       title 'saslib.rcp';


data saslib.xns;
                    set putget;
                    if prot   'XNS';
proc print data   saslib.xns;
                       title 'saslib.xns';


data saslib.nfs;
                    set putget;
                    if prot   'NFS';
proc print data   saslib.nfs;
                       title 'saslib.nfs';
```

SAS Procedure #2, which produces 6 plots:

```
/*********************************************************
Name:  gplots.sas

Description:
gplots.sas uses the permanent SAS data sets created by
getdata.sas and generates 7 plots, illustrating file
transfer protocol implementation performance as a
function of filesize and transfer direcion:

1)  all 5 protocols writing
2)  all 5 protocols reading
3)  FTAM read vs. write
4)  FTP read vs. write
5)  UNIX rcp read vs. write
6)  XNS read vs. write
7)  NFS read vs. write

The horizontal axis on all the plots represents filesize in bytes.
The vertical axis on all the plots represents seconds.

Maintenance:
4/91              Elayne McFaul Original Author
*********************************************************/

libname  saslib  '[mcfaul11.sas.thesis]';

goptions device=manplot nocharacters ;

symbol1 color=black i=join v=M;
symbol2 color=black i=join v=F;
symbol3 color=black i=join v=R;
symbol4 color=black i=join v=X;
symbol5 color=black i=join v=N;

title1 j=c h=2 f=triplex '          Performance of Filing Protocols' ;
```

```
title2 j=c h=2 f=triplex '          Client writing to Server' ;
footnote1 j=r h=1 f=triplex 'M=FTAM  F=FTP  R=rcp  X=XNS  N=NFS';
footnote2 ' ';
footnote3 j=c h=1 f=triplex '          Figure 15';
footnote4 ' ';
footnote5 j=c h=1 f=triplex '          -53-';

proc gplot data   saslib.allputs;
axis1             order=0 to 15 by 1
                  label=(f=triplex h=1 j=c '          Seconds')
                  value=(f=triplex h=1);

axis2             order=1,200000,400000,600000,800000,1000000
                  label=(f=triplex h=1 j=c 'Filesize in bytes')
                  value=(f=triplex h=1);

                  plot ftam*filesize=1 ftp*filesize=2
                       rcp*filesize=3 xns*filesize=4
                       nfs*filesize=5 /overlay
                  vaxis   axis1
                  vzero
                  haxis   axis2;
run;


title1 j=c h=2 f=triplex '          Performance of Filing Protocols' ;
title2 j=c h=2 f=triplex '          Client reading from Server' ;
footnote3 j=c h=1 f=triplex '          Figure 16';
footnote4 ' ';
footnote5 j=c h=1 f=triplex '          -54-';
proc gplot data   saslib.allgets;
                  plot ftam*filesize=1 ftp*filesize=2
                       rcp*filesize=3 xns*filesize=4
                       nfs*filesize=5 /overlay
                  vaxis   axis1
                  vzero
                  haxis   axis2;
```

```
run;



symbol1 color=black i=join v=W;
symbol2 color=black i=join v=R;
title1 j=c h=2 f=triplex '          Symmetry of Filing Protocols' ;
title2 j=c h=2 f=triplex '          Protocol   FTAM';

footnote1 j=r h=1 f=triplex 'R=Client Reading  W=Client Writing';
footnote2 ' ';
footnote3 j=c h=1 f=triplex '          Figure 17';
footnote4 ' ';
footnote5 j=c h=1 f=triplex '          -55-';

proc gplot data   saslib.ftam;
                  plot put*filesize=1 get*filesize=2 /overlay
                  vaxis   axis1
                  vzero
                  haxis   axis2;
run;



title1 j=c h=2 f=triplex '          Symmetry of Filing Protocols' ;
title2 j=c h=2 f=triplex '          Protocol   FTP';
footnote3 j=c h=1 f=triplex '          Figure 18';
footnote4 ' ';
footnote5 j=c h=1 f=triplex '          -56-';

proc gplot data   saslib.ftp;
                  plot put*filesize=1 get*filesize=2 /overlay
                  vaxis   axis1
                  vzero
                  haxis   axis2;
run;
```

```
title1 j=c h=2 f=triplex '          Symmetry of Filing Protocols' ;
title2 j=c h=2 f=triplex '          Protocol   UNIX rcp';
footnote3 j=c h=1 f=triplex '          Figure 19';
footnote4 ' ';
footnote5 j=c h=1 f=triplex '          -57-';

proc gplot data   saslib.rcp;
                  plot put*filesize=1 get*filesize=2 /overlay
                  vaxis   axis1
                  vzero
                  haxis   axis2;
run;


title1 j=c h=2 f=triplex '          Symmetry of Filing Protocols' ;
title2 j=c h=2 f=triplex '          Protocol   XNS';
footnote3 j=c h=1 f=triplex '          Figure 20';
footnote4 ' ';
footnote5 j=c h=1 f=triplex '          -58-';

proc gplot data   saslib.xns;
                  plot put*filesize=1 get*filesize=2 /overlay
                  vaxis   axis1
                  vzero
                  haxis   axis2;
run;


title1 j=c h=2 f=triplex '          Symmetry of Filing Protocols' ;
title2 j=c h=2 f=triplex '          Protocol   NFS';
footnote3 j=c h=1 f=triplex '          Figure 21';
footnote4 ' ';
footnote5 j=c h=1 f=triplex '          -59-';

proc gplot data   saslib.nfs;
                  plot put*filesize=1 get*filesize=2 /overlay
                  vaxis   axis1
```

```
                    vzero
                    haxis   axis2;
run;
```

Appendix C

Source code segments for the port of XNS Filing from a SPP transport to a TCP transport.

Most of the code below employs the *#ifdef* preprocessor check command to determine whether the COURIER_TCP constant has been set. If set, a TCP transport is desired. Otherwise a SPP transport is used.

Below, all calls to the Clearinghouse service were replaced with calls to the *gethostbyname* library function. Also, all XNS-formatted addresses were replaced with an Internet-formatted address.

```
#ifndef COURIER←TCP

  if ((hostaddr   CH←LookupAddrDN( hostobjname, 0, hnamebuf, 128))) {
      /* should check here to be sure host is a file service */
      hostaddr->x←port   htons(5); /* ?? */
      cconn   CourierOpen(hostaddr);
      if ( cconn == (CourierConnection *) 0 ) {
          connerr.problem= FilingSubset1←noResponse;
          raise(FilingSubset1←ConnectionError, &connerr);
      }
      /* reset objname to flush wildcards */
      /* clear←Clearinghouse3←ThreePartName(&hostobjname); */
      hostobjname   CH←StringToName(hnamebuf, &defaultobjname);
      hostname   hnamebuf;
      if (verbose)
        printf("Connected to %s\n", hnamebuf);
  } else {
      printf("%s: unknown host\n", name);
      usefiling= 1;
      cconn   (CourierConnection*)0;
  }
#else  /* COURIER←TCP */
if ((hp   gethostbyname (name)) == 0) {
```

```
            printf("%s: unknown host\n", name);
        usefiling= 1;
        cconn   (CourierConnection*)0;
    }
    destaddr   (struct in←addr *) malloc (25);
    bcopy (hp->h←addr, destaddr, hp->h←length);
    cconn   CourierOpen(destaddr);
    if ( cconn == (CourierConnection *) 0 ) {
        connerr.problem= FilingSubset1←noResponse;
        raise(FilingSubset1←ConnectionError, &connerr);
    }
    if (verbose)
      printf("Connected to %s\n", name);

#endif COURIER←TCP
```

The version of Courier used in this study was built to use the Sequenced Packet Convergence Protocol (SPCP). SPCP is a "shim protocol" that provides an SPP abstraction over TCP. Below, the SPP socket calls are replaced with TCP socket calls.

```
#ifndef COURIER←TCP

CourierConnection *
CourierOpen( addr )
   struct ns←addr *addr;
{
   extern char *malloc();
   CourierConnection *conn;

   conn   (CourierConnection*) malloc(sizeof(CourierConnection));
   conn->host.sns←family   AF←NS;
   conn->host.sns←addr   *addr;
   /* unknown socket? */
   if (conn->host.sns←addr.x←port == 0)
       conn->host.sns←addr.x←port   htons(IDPPORT←COURIER);
```

```
#if DEBUG

  if (CourierClientDebuggingFlag)
      fprintf(stderr,
          "[CourierOpen: connect to %x#%x.%x.%x.%x.%x.%x#%x]\n",
          ns←netof(conn->host.sns←addr),
          conn->host.sns←addr.x←host.c←host[0],
          conn->host.sns←addr.x←host.c←host[1],
          conn->host.sns←addr.x←host.c←host[2],
          conn->host.sns←addr.x←host.c←host[3],
          conn->host.sns←addr.x←host.c←host[4],
          conn->host.sns←addr.x←host.c←host[5],
          ntohs(conn->host.sns←addr.x←port));

#endif

  if ((conn->fd   openSPPConnection(&conn->host)) >= 0) {
      conn->abortseen   FALSE;
      conn->bdtstate   wantdata;
      conn->state   wantversion;
      conn->sphdrOpts.sp←dt   0;
      conn->sphdrOpts.sp←cc   0;
      conn->begin = conn->end   conn->buf;
      return(conn);
  }
  else {
      free((char*)conn);
      return(NULL);
  }
}


#else /* COURIER←TCP */

CourierConnection *
CourierOpen (addr)
```

```
    struct in←addr *addr;
{
    extern char *malloc ();
    CourierConnection *conn;

    conn    (CourierConnection*) malloc (sizeof (CourierConnection));
    conn->host.sin←family    AF←INET;
    conn->host.sin←addr    *addr;

    if ((conn->fd    openTCPConnection (&conn->host)) >= 0) {
        conn->abortseen    FALSE;
        conn->bdtstate    wantdata;
        conn->state    wantversion;
        conn->begin = conn->end    conn->buf;
        return(conn);
    }
    else {

        free ((char*)conn);
        return (NULL);
    }
}  /* CourierOpen */

#endif COURIER←TCP
```

Below, all calls to the SPP-based Authentication service were commented out.

```
#ifndef COURIER←TCP
        if ( !Auth←CredCheck(user←credentials, user←verifier) ) {
            ReturnAuthenticationError(AUTHENTICATION←credentialsInvalid);
            /* NOT REACHED */
        }
#endif COURIER←TCP
```

```
#ifndef COURIER←TCP
   if (user←credentials.primary.type == AUTHENTICATION←simpleCredentials)
{
       if ( !Auth←CredCheck(user←credentials.primary, user←verifier) ) {
           ReturnAuthenticationError(FILING←primaryCredentialsInvalid);
           /* NOT REACHED */
       }
   }

   if ( get←name←and←pwd(&user←credentials.secondary, user, pass) != -1 )
{
       ReturnAuthenticationError(FILING←secondaryCredentialsRequired);
       /* NOT REACHED */
   }
#endif COURIER←TCP
```